



LARAVELISTA

A COLLECTION OF
LARAVEL TUTORIALS

MARIO BAŠIĆ

A Collection of Laravel Tutorials

Mario Bašić

This book is for sale at <http://leanpub.com/laravelista-collection>

This version was published on 2018-05-02



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2017 - 2018 Mario Bašić

Contents

About the Author	i
About the Book	ii
Getting Started	1
Elementary Laravel	2
Laravel on Windows with Homestead	29
Laravel on Windows with Laragon	51
Configure Laravel 5 for Shared Hosting	64
The HTTP Layer	72
Handling Nested Resources	73
Route Caching in Laravel	79
Laravel Forms & HTML	81
Different ways of validating requests	95
Validating Form Arrays	97
Creating Custom Validation Rules	101
Frontend: Compiling Assets	104
Sublimely Magnificent Laravel Mix	105

CONTENTS

Laravel Mix Without Laravel	112
Laravel Elixir Without Laravel	118
General Topics	126
Sitemap for better SEO	127
RSS feed for news readers	131
Newsletter subscription with MailChimp	135
PHPUnit Code Coverage Analysis	147
Creating a Sitemap with Bard 2.0	154
Bard 2.0 and Laravel	160
Multilingual Web Application with Laravel	164
API Development	201
Laravel API 101	202
Write better API documentation with API Blueprint	268
Database & Eloquent ORM	282
Paginating a collection of different models	283
MySQL ON DELETE CASCADE or SET NULL	290
Dropping a composite primary key on a pivot table	292
Understanding Pagination Presenters	295
Testing	300
Laravel 5 with Codeception	301
Lumen 5 with Codeception	307

CONTENTS

Deployment	313
Deploy Laravel with Envoy	314
Advanced Laravel Envoy	319
Deploying a Laravel App from GitHub to Heroku	326
 Server Administration	 344
Bash Scripting Introduction	345
Upgrade to PHP 7 on Ubuntu 14.04	349
Secure Nginx with Let's Encrypt on Ubuntu 14.04	352
Using SSH to execute commands on a remote server	357
 Extras	 363
JSON Web Token Authentication for Lumen REBOOT	364
What is a canonical tag?	373
Loading your own fork of a third party library	376
The End	380

About the Author

Web developer, Laravel enthusiast, IT manager and soon to be father.

About the Book

This book contains all tutorials that have been published on the Laravelista website in the period from ≥ 2016 . to < 2018 .

A programmer guides an another programmer to build stuff.

This book represents my two year effort to teach others how to build web sites and web applications using Laravel framework. It is a great starting point if you want to jump straight to the thing that interests you.

All tutorials in this book come with a working code repository on GitHub. Throughout the whole book you will find tips to expand your knowledge on a topic. Each chapter comes with a commit that shows the changes that have been made in it. No need to read the whole book. Find a tutorial that interests you and start reading.

- Building and documenting an API with Laravel
- Most popular tutorial on JWT for Lumen
- Homestead or Laragon, pick one and start building
- Testing with Codeception on Laravel and Lumen

Forms and Validation

Learn how to create and validate forms. Includes tutorials on validating form arrays and different ways of validating data.

Compiling Frontend Assets

Learn how to use Laravel Mix to compile frontend assets. Also, learn how to use Laravel Mix or Laravel Elixir without Laravel.

Multilingual Web Applications

Learn how to create multilingual web application with Laravel. From simple text localization to database model translations.

Deployment with Envoy

Learn everything about Laravel Envoy with practical examples. There is also a tutorial on deploying a Laravel app from GitHub to Heroku.

Database and Eloquent ORM

Learn all about pagination presenters and how to paginate a collection of different models. Also, learn different ways of deleting records.

Server Administration

For the brave, there are a couple of tutorials on installing Nginx, PHP7, obtaining certificates with Let's encrypt and bash scripting.

Getting Started

Homestead or Laragon, pick one and start building.

Elementary Laravel

Only the basics in the simplest way possible. Everything you need to create a website.



View Source Code

Source code for this tutorial is available [here](#)¹.

Installation

We have to install Laravel before we can start using it, so let's do that now.

Published at: 17. October, 2016.

Welcome to my new course called *Elementary Laravel*. In this course, we will build a simple *business* website with Laravel 5.3. The idea is to teach you how to build a website with Laravel by using the least amount of steps necessary and provide you with references to expand the knowledge you obtain from this course.

Don't bother yourself with questions like: "Is this the best practice?" or "Should I be doing this-this way?" etc. Everybody starts somewhere and I think that this is the best starting point for learning Laravel. By the end of this course, you will have a working *business* website and basic knowledge about Laravel.

Requirements

These are the tools that you will need for this tutorial:

- [PHP](#)²
- [Composer](#)³
- A text editor or an IDE. *I suggest using [Atom](#)*⁴.
- A Web browser. *[Chrome](#)*⁵ preferred.

Installation

If you haven't already, download the Laravel installer using Composer:

¹<https://github.com/laravelista/elementary-laravel>

²<http://php.net/>

³<https://getcomposer.org/>

⁴<https://atom.io/>

⁵<https://www.google.com/chrome/index.html>

Installing Laravel Installer

```
composer global require "laravel/installer"
```

To create a fresh Laravel installation we will use this command:

Creating a fresh Laravel installation

```
laravel new website
```

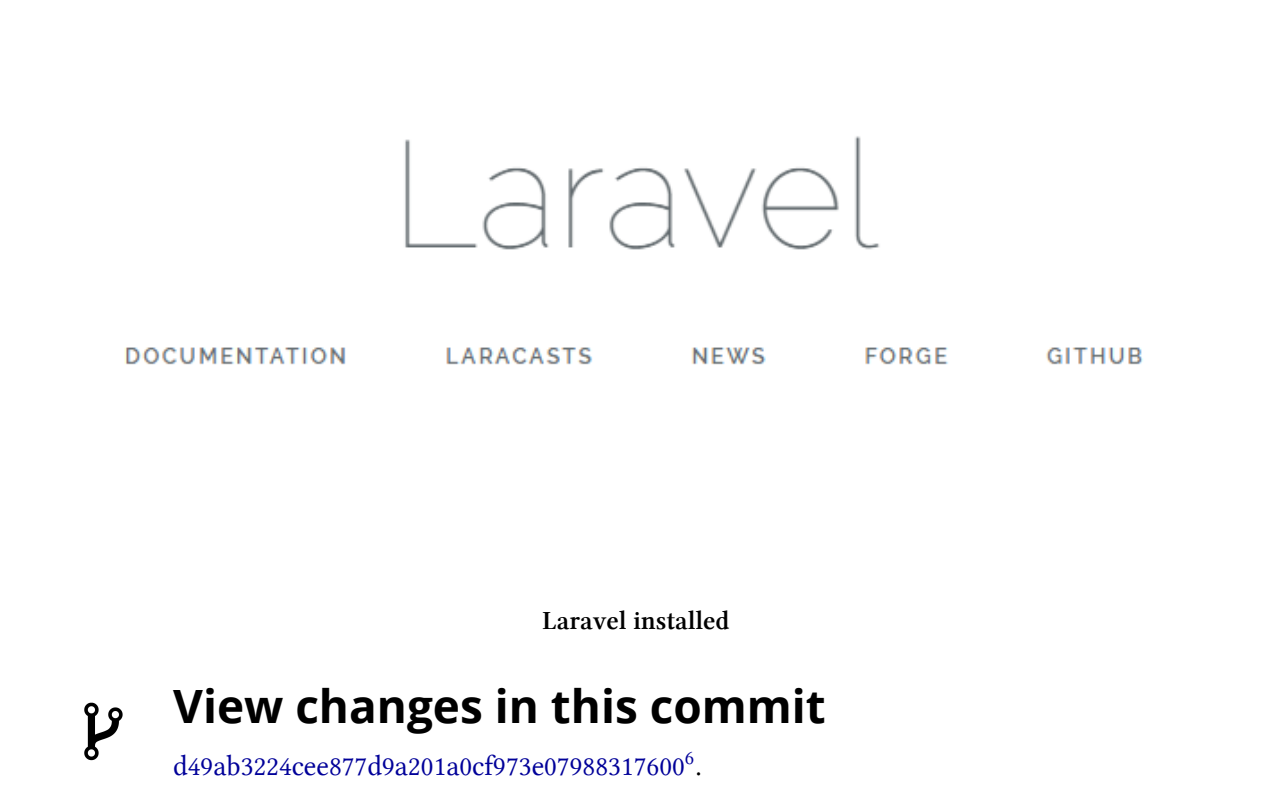
This command will create a directory called `website` containing a fresh Laravel installation.

To see how our newly installed Laravel application looks like we will use the built-in Local development server. To start a development server at `http://localhost:8000` run this command:

Starting a development server

```
php artisan serve
```

Now open your browser to that URL and you will see this screen:



⁶<https://github.com/laravelista/elementary-laravel/commit/d49ab3224cee877d9a201a0cf973e07988317600>



Improve your skills!

Instead of using the built-in PHP development server, you should really learn how to install and configure [Homestead](#)⁷. If you are on Windows, I already have a course on [Homestead on Windows](#) and [Laragon on Windows](#). If you are on a Mac, take a look at [Valet](#)⁸.

You now have Laravel installed and ready to go.

Routing

Routes are the entry points to your application, so it is only logical to learn about them first.

Published at: 17. October, 2016.

Routes are the main entry points for your Laravel application. With routes, we define URLs that are accessible on our website.

If you need an about page, you probably want it to be accessible at /about URL. Routes define what URLs are accessible and what happens when a route is triggered. Think of them as an index for your website. When you want to locate something, you just have to take a look at the routes file.

URL structure

For our *business* website, we will have a structure like this:

- GET / - Our home page.
- GET /about - The about page.
- GET /contact - Contact page that has a contact form.
- POST /contact - When contact form is submitted, data will be sent to this route.

If you don't know already what GET and POST mean, think of them this way. GET is used for getting a web page. POST is used to send data to a web page. While this isn't the exact definition, for the purpose of this tutorial it will do.



Improve your skills!

If you are thinking of starting a career in web development, you should learn more about HTTP methods: [Method definitions](#)⁹, [HTTP Methods: GET vs. POST](#)¹⁰ and [Using HTTP Methods for RESTful Services](#)¹¹.

⁷<https://laravel.com/docs/5.3/homestead>

⁸<https://laravel.com/docs/5.3/valet>

⁹<https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html>

¹⁰http://www.w3schools.com/tags/ref_httpmethods.asp

¹¹<http://www.restapitutorial.com/lessons/httpmethods.html>

Defining our routes

To define the routes for our website, in your text editor open the routes file `routes/web.php`.

You should see the default route which you can see if you visit `http://localhost:8000` in your browser and a comment describing this routes file.

Since we want to keep the route `/` for our home page, we only need to change the view that is being returned from that route closure. Change the line `return view('welcome');` to `return view('home');`.

Now to create the rest of our routes, add this below that route:

Adding routes to the routes file

```
Route::get('about', function() {
    return view('about');
});

Route::get('contact', function() {
    return view('contact');
});

Route::post('contact', function() {
    //
});
```

Save the changes and open your browser to `http://localhost:8000`. You should get an error now saying `View [home] not found..` This means that Laravel has not found the home view file that we specified in our `/` route. Our next step is to create that view file and any other view file that we specified in our routes file.



View changes in this commit

[4f504843665c9e1d0cd8d41e0eb60259daaeaabc](https://github.com/laravelista/elementary-laravel/commit/4f504843665c9e1d0cd8d41e0eb60259daaeaabc)¹².

Views

Views contain the HTML served by your application and separate your application logic from your presentation logic.

¹²<https://github.com/laravelista/elementary-laravel/commit/4f504843665c9e1d0cd8d41e0eb60259daaeaabc>

Published at: 24. October, 2016.

Views are files located in `resources/views` with `.blade.php` extension. Views contain HTML which is served by your application.

For our *business* website we need to create the views that we have specified in our routes file.

Create View Files

Create the following files in the `resources/views` directory:

- `home.blade.php`
- `about.blade.php`
- `contact.blade.php`

If you try to view the home page now, you will see a blank page. That is also true for all other routes that we have defined.

Bootstrap Home Page

Good, now we will add HTML to our `resources/views/home.blade.php` view. We will be using [Bootstrap](http://getbootstrap.com)¹³ to quickly get started with some basic page design. Open the file and add the following inside it:

Bootstrapping Bootstrap

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Home page</title>

    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7\
/css/bootstrap.min.css" integrity="sha384-BVYiISIFeK1dGmJRAkycuHAHRg320mUcww7on3\
RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
  </head>
  <body>
    <h1>Hello, world!</h1>
```

¹³<http://getbootstrap.com>

```

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.\
js"></script>
<!-- Latest compiled and minified JavaScript -->
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.mi\
n.js" integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIPg9mGCD8wGN\
IcPD7Txa" crossorigin="anonymous"></script>
</body>
</html>

```

Save the changes and open your browser to /. Hit F5 to refresh the page if needed. You should see Hello, world! displayed on the page. **Excellent!** Now you can use Bootstrap on this page to create your home page.



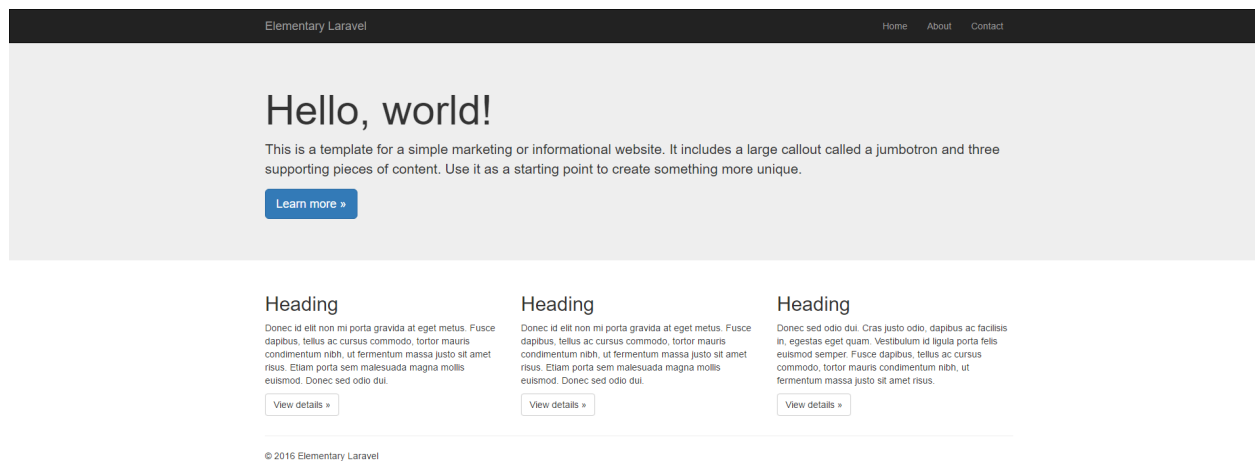
Improve your skills!

Bootstrap is the fastest way to get started with a new website. It contains almost everything needed to design a website and it also looks good by default. I suggest that you read: *Getting started*, *CSS* and *Components* pages from its [website](http://getbootstrap.com/)¹⁴.

This is how I have made the home page to look using Bootstrap. I have used a [Jumbotron](http://getbootstrap.com/examples/jumbotron/)¹⁵ example as a starting point.

¹⁴<http://getbootstrap.com/>

¹⁵<http://getbootstrap.com/examples/jumbotron/>



Home page

Feel free to change the page however you like it.



View changes in this commit

[8a5128154e3cc0db6f02780029538ca3ec18f52b](https://github.com/laravelista/elementary-laravel/commit/8a5128154e3cc0db6f02780029538ca3ec18f52b)¹⁶.

Use Helpers

There are a few things that we can do to improve our page. First, if you look at the [home.blade.php file](#)¹⁷ you can notice that for defining links to our other routes we use `About`. While this will work, it is much better to use a Laravel helper function `url('/about')` which creates a full URL to the route.

Change all links so that they use the Laravel `url` helper. For example, change `Home` to `Home`. Do this for all links.



View changes in this commit

[2df2d0ae2172affda92c9142753c44814b651752](https://github.com/laravelista/elementary-laravel/commit/2df2d0ae2172affda92c9142753c44814b651752)¹⁸.

Before you start copying the HTML from our home page to other pages, ask yourself *Is there a way to reuse sections of our homepage on other pages to avoid copying the code?*

¹⁶<https://github.com/laravelista/elementary-laravel/commit/8a5128154e3cc0db6f02780029538ca3ec18f52b>

¹⁷<https://github.com/laravelista/elementary-laravel/blob/8a5128154e3cc0db6f02780029538ca3ec18f52b/resources/views/home.blade.php>

¹⁸<https://github.com/laravelista/elementary-laravel/commit/2df2d0ae2172affda92c9142753c44814b651752>

As you may have noticed `{{ }}` is used to echo a value in the view. There will be more talk about this in the next tutorial.

Blade templates

Blade is a templating engine provided with Laravel and unlike other templating engines it does not restrict you from using plain PHP code in your views.

Published at: **02. November, 2016.**

In my opinion one of the best parts of Laravel is the Blade templating engine. It already comes with Laravel and has everything you need and more. It enables you to work with templates and layouts, display data, use control structures, include subviews, use stacks, inject services in views and if that is still not enough you can easily extend it to do whatever you desire.

The purpose of this tutorial is not to teach you everything that Blade can do, but to teach you about templates, layouts, sections, subviews and basic data presentation.



Improve your skills!

Learn more about Blade templates by reading the [documentation](https://laravel.com/docs/5.3/blade)¹⁹. It is very important to know what you can do with it.

In our previous tutorial I have left you with a question to ask yourself: *Is there a way to reuse sections of our home page on other pages to avoid copying the code?*

The answer is yes, there is and it is called Blade templates.

Layouts

When building your templates you should start from the most outer shell and those are the HTML tags `html`, `head` and `body`. We will extract a part of the HTML from our `resources/views/home.blade.php` file that is common for all other pages and place it in `resources/views/layouts/default.blade.php`.

What I want you to do now is to move this:

¹⁹<https://laravel.com/docs/5.3/blade>

Extracting a layout page

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Home page</title>

    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7\
/css/bootstrap.min.css" integrity="sha384-BVYiISIFeK1dGmJRAkycuHAHRg32OmUcww7on3\
RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
    <style>
      body {
        padding-bottom: 20px;
      }
      .navbar {
        margin-bottom: 0px;
        border-radius: 0;
      }
    </style>
  </head>
  <body>

    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.\
js"></script>
    <!-- Latest compiled and minified JavaScript -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.mi\
n.js" integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGN\
IcPD7Txa" crossorigin="anonymous"></script>
  </body>
</html>

```

to a new file in resources/views/layouts/default.blade.php. So that you home.blade.php file now only contains:

Content of the home page file

```

<nav class="navbar navbar-inverse">
  <div class="container">
    <div class="navbar-header">
      <button type="button" class="navbar-toggle collapsed" data-toggle="collaps\
e" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
        <span class="sr-only">Toggle navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>
      <a class="navbar-brand" href="{{ url('/') }}">Elementary Laravel</a>
    </div>
    <div id="navbar" class="navbar-collapse collapse">
      <ul class="nav navbar-nav navbar-right">
        <li><a href="{{ url('/') }}">Home</a></li>
        <li><a href="{{ url('/about') }}">About</a></li>
        <li><a href="{{ url('/contact') }}">Contact</a></li>
      </ul>
    </div><!--/.navbar-collapse -->
  </div>
</nav>

<!-- Main jumbotron for a primary marketing message or call to action -->
<div class="jumbotron">
  <div class="container">
    <h1>Hello, world!</h1>
    <p>This is a template for a simple marketing or informational website. It in\
cludes a large callout called a jumbotron and three supporting pieces of content\
. Use it as a starting point to create something more unique.</p>
    <p><a class="btn btn-primary btn-lg" href="#" role="button">Learn more »</a>\
</p>
  </div>
</div>

<div class="container">
  <!-- Example row of columns -->
  <div class="row">
    <div class="col-md-4">
      <h2>Heading</h2>
      <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus\
ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit\

```

```

amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.\
</p>
    <p><a class="btn btn-default" href="#" role="button">View details »</a></p>
</div>
<div class="col-md-4">
    <h2>Heading</h2>
    <p>Donec id elit non mi porta gravida at eget metus. Fusce dapibus, tellus\
ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit\
amet risus. Etiam porta sem malesuada magna mollis euismod. Donec sed odio dui.\
</p>
    <p><a class="btn btn-default" href="#" role="button">View details »</a></p>
</div>
<div class="col-md-4">
    <h2>Heading</h2>
    <p>Donec sed odio dui. Cras justo odio, dapibus ac facilisis in, egestas e\
get quam. Vestibulum id ligula porta felis euismod semper. Fusce dapibus, tellus\
ac cursus commodo, tortor mauris condimentum nibh, ut fermentum massa justo sit\
amet risus.</p>
    <p><a class="btn btn-default" href="#" role="button">View details »</a></p>
</div>
</div>

<hr>

<footer>
    <p>© 2016 Elementary Laravel</p>
</footer>
</div> <!-- /container -->

```

Now that we have created our first layout template in `layouts/default.blade.php`, we must tell our view `home.blade.php` to extend upon that layout. We do that by adding

Extending a page with a layout

```
@extends('layouts.default')
```

at the top of the file `home.blade.php`. There is one more step before we continue. We have to tell our layout where to display the HTML from the page we want.

Sections

Sections are used to tell the layout where we want the content of a section to be displayed.

Go to our `layouts/default.blade.php` file and just below the opening body tag place the following:

Defining a section for content

```
@yield('content')
```

and now in `home.blade.php` wrap all content below `@extends('layouts.default')` in a section block:

Populating content section

```
@section('content')
    {{-- Place wrapped content instead of this Blade comment --}}
@stop
```

If you take a look at `http://localhost:8000` you should see that everything looks the same. That is the point, but can you notice how much cleaner our home view looks now?

Subviews

We can make it even cleaner by using subviews to extract our navigation to a subview which we will include in our default layout template.

Create a new file in `resources/views/layouts/partials/navbar.blade.php` and move our navbar from `home.blade.php` to that file.

Creating a navbar partial

```
<nav class="navbar navbar-inverse">
    <div class="container">
        <div class="navbar-header">
            <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
                <span class="sr-only">Toggle navigation</span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
                <span class="icon-bar"></span>
            </button>
            <a class="navbar-brand" href="{{ url('/') }}">Elementary Laravel</a>
        </div>
        <div id="navbar" class="navbar-collapse collapse">
            <ul class="nav navbar-nav navbar-right">
                <li><a href="{{ url('/') }}">Home</a></li>
                <li><a href="{{ url('/about') }}">About</a></li>
                <li><a href="{{ url('/contact') }}">Contact</a></li>
```

```

        </ul>
    </div><!--/.navbar-collapse -->
</div>
</nav>

```

Good, now we have to tell our layout to include that subview. Go to `layouts/default.blade.php` and just above `@yield('content')` place the following:

Including navbar partial

```
@include('layouts.partials.navbar')
```

Save the changes and if you look at the browser and hit refresh it should still look the same, but our home view file is even cleaner now. Awesome!

There are more things that you can move into subviews if you think that you can benefit from it, but for this tutorial, this seems fine to me.

Data presentation

The name of this chapter is misleading at best, I know.

If you have been following along, you may have noticed that in our default layout file the `title` tag is hardcoded to be `Home page`, but what about our other pages? Should the title bar not hold some other value instead of `Home page`?

There is a simple way of achieving this. Go to `layouts/default.blade.php` and replace the value of `title` to be:

Defining a place for title section

```
@yield('title', 'Elementary Laravel')
```

This tells our template to place a section from our view called `title` here, but if it cannot find it then place the default value of `Elementary Laravel`.

Now to set the title in our view file, go to `home.blade.php` and create a new section just below the `@extends('layouts.default')` and above the `@section('content')`:

Creating a title section

```
@section('title', 'Home page')
```

If you take a look at <http://localhost:8000> you should see that everything looks the same. Great job!

Wrapping things up

Now we have a way to populate our other views `about.blade.php` and `contact.blade.php`. Copy the following to those files and change the title accordingly:

Page skeleton

```
@extends('layouts.default')

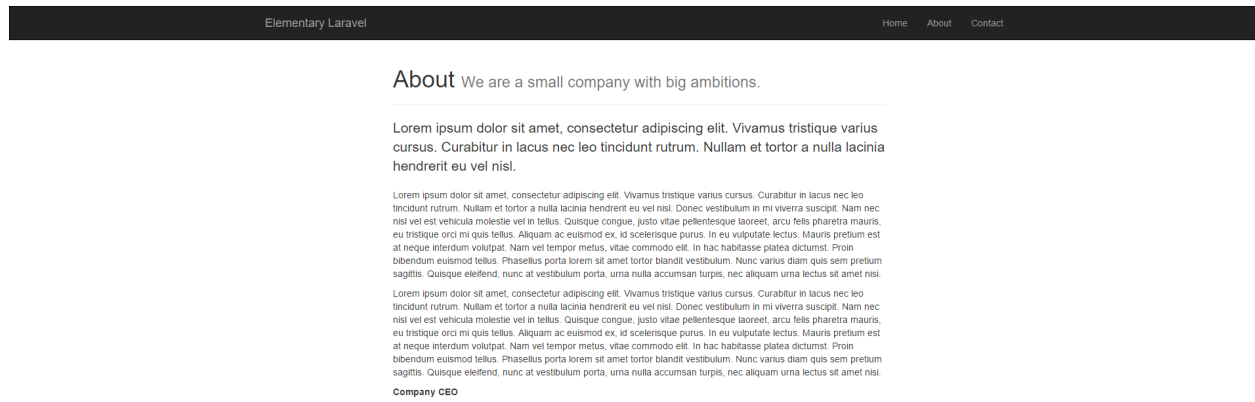
@section('title', 'About page')

@section('content')
    {{-- Place content instead of this Blade comment --}}
@stop
```

I will quickly populate those pages with some Bootstrap. You can modify them however you want or you can copy the code from what I have done:

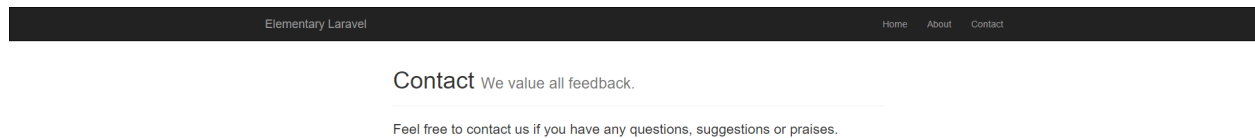
[about.blade.php](#)²⁰

²⁰<https://raw.githubusercontent.com/laravelista/elementary-laravel/00f0293ad4668f08f11ba67c890f892bddb579d8/resources/views/about.blade.php>



About page

`contact.blade.php`²¹



Contact page

²¹<https://raw.githubusercontent.com/laravelista/elementary-laravel/00f0293ad4668f08f11ba67c890f892bddd579d8/resources/views/contact.blade.php>



View changes in this commit

[00f0293ad4668f08f11ba67c890f892bddb579d8](https://github.com/laravelista/elementary-laravel/commit/00f0293ad4668f08f11ba67c890f892bddb579d8)²².

We now have a home and about pages complete. On our contact page, we surely want to have a contact form, which sends an email upon successful validation.

Forms

Forms are an important part of any web application. I'll show you a quick and easy way to create a form with helpers.

Published at: 11. November, 2016.

We will now create a contact form. We will use normal HTML with some Laravel helpers in order to provide a better user experience.



Improve your skills!

To improve upon the form that we will create in this tutorial I advise you to take a look at [Forms & HTML](#)²³ package. I have an in-depth tutorial about it called [Laravel Forms & HTML](#) so be sure to check it out.

Create the form

We will use basic Bootstrap styling to design the form. We will require from the user to enter his:

- name
- email
- message (*comment* - We will cover at a later point)



Improve your skills!

It is very important to know how to build and design forms with Bootstrap, so I recommend reading the [documentation](#)²⁴ about it.

In our `contact.blade.php` you will see a Blade comment `{{-- Contact form goes here --}}`. Replace that line with the following:

²²<https://github.com/laravelista/elementary-laravel/commit/00f0293ad4668f08f11ba67c890f892bddb579d8>

²³<https://laravelcollective.com/docs/5.3/html>

²⁴<http://getbootstrap.com/css/#forms>

Creating a contact form

```
<form method="POST" action="{{ url('/contact') }}">
    {{ csrf_field() }}
    <div class="form-group">
        <label for="name">Name</label>
        <input id="name" type="text" class="form-control" name="name" value="{{ \
old('name') }}" placeholder="Your name">
    </div>
    <div class="form-group">
        <label for="email">E-mail</label>
        <input id="email" type="email" class="form-control" name="email" value="{{ \
{{ old('email') }}" placeholder="Your E-mail">
    </div>
    <div class="form-group">
        <label for="comment">Message</label>
        <textarea rows="10" id="comment" class="form-control" name="comment" pla\
ceholder="Your message">{{ old('comment') }}</textarea>
    </div>
    <button type="submit" class="btn btn-primary btn-lg">Send</button>
</form>
```

I will explain the helpers used here in the following chapter, but for now, save the changes and open your browser to <http://localhost:8000/contact>. You will see that the page looks like this now:

Elementary Laravel
Home
About
Contact

Contact

We value all feedback.

Feel free to contact us if you have any questions, suggestions or praises.

Name

E-mail

Message

Send

Contact with form

We have specified that we want to POST the data from the form to /contact URL. If you remember our routes file from a few tutorials ago, we have a route for that method:

Adding a route for posting contact form

```
Route::post('contact', function() {
    //
});
```

If you press Send on the form now, you will get a blank page. That is because we are not returning anything from our route that handles form submission.



View changes in this commit

[315057e9e97d0d34cde9a683cd00bd5e2dedfdff](https://github.com/laravelista/elementary-laravel/commit/315057e9e97d0d34cde9a683cd00bd5e2dedfdff)²⁵.

Helpers used

We have used two Laravel helpers in this form.



Improve your skills!

To learn more about all helpers that come with Laravel visit the documentation for [Helpers](https://laravel.com/docs/5.3/helpers)²⁶.

old()

The `old` function retrieves an old input value flashed into the session. This will be very helpful in our next tutorials where we will tackle validation. What this does is it keeps the value that the user entered in the input field so that if the validation fails, the input entered by the user is preserved. He does not need to type it again.

csrf_field()

The `csrf_field` function generates an HTML hidden input field containing the value of the CSRF token. Laravel automatically generates a CSRF “token” for each active user session managed by the application. This token is used to verify that the authenticated user is the one actually making the requests to the application. Read more about this [here](https://laravel.com/docs/5.3/csrf)²⁷.

A quick recap of this tutorial:

²⁵<https://github.com/laravelista/elementary-laravel/commit/315057e9e97d0d34cde9a683cd00bd5e2dedfdff>

²⁶<https://laravel.com/docs/5.3/helpers>

²⁷<https://laravel.com/docs/5.3/csrf>

- I've shown you how to build a contact form
- You have learned about `old` and `csrf_field` helpers
- You have been given a lot of documentation to read in order to improve your skills
- The form can be submitted and we are returned a blank screen

What we are still missing is *Validation*. Are we going to blindly believe our users, that they have entered a valid email address or that they have entered all fields that we require? Hell no! This is where Laravel shines, the **Validation**.

Validation

Validation consists of two parts, validating the data from the user and displaying the errors messages back to the user..

Published at: 28. November, 2016.

Out of the box, Laravel comes loaded with [validation options](#)²⁸ and it is also very easy and quick to implement it however you want. In the previous tutorial, we have left things off at a contact form. We have created a contact form which submits its data to the URL we specified `POST /contact` and it is being handled by our route.

Scenarios

This is what we want to happen when the user submits the contact form:

If the data entered passes validation, our application should:

- send us an email
- redirect the user to `/contact` page
- display success message to the user

If the data entered does not pass validation, our application should:

- redirect the user to the contact form **with old input**
- display errors messages telling the user what he did wrong

Validating data

Back to our `routes/web.php` file. Locate the route:

²⁸<https://laravel.com/docs/5.3/validation>

Empty contact POST route

```
Route::post('contact', function() {
    //
});
```

First we have to tell our route to use dependency injection to inject the Request like so:

Injecting Request dependency

```
use Illuminate\Http\Request;
Route::post('contact', function(Request $request) {
    // place code here
});
```

Now we can access the request, meaning that we can validate the data inside it. We will manually build our Validator but we will also use the *Automatic Redirection* feature to automatically handle the redirection and error processing if the validation fails.

Place this code inside our POST route:

Validating Request data

```
\Validator::make($request->all(), [
    'name' => 'required|string',
    'email' => 'required|email',
    'comment' => 'required|string'
])->validate();

// normal code execution with successful validation.
// send email or do whatever you want here,
// redirect user back and notify him of our success
```

Now let me explain. From the [Validation documentation](#)²⁹:

“If you would like to create a validator instance manually but still take advantage of the automatic redirection offered by the `ValidatesRequest` trait, you may call the `validate` method on an existing validator instance. If validation fails, the user will automatically be redirected or, in the case of an AJAX request, a JSON response will be returned.”

As you can see, we are specifying parameter names and validation rules for each.

²⁹ <https://laravel.com/docs/5.3/validation#automatic-redirection>



Improve your skills!

To understand and know which other validation rules exist, read the documentation on [Available Validation Rules](#)³⁰.



View changes in this commit

[dbc88e6ccae9f8c8b0c92e9b19edb7e3cefd8949](#)³¹.

Displaying errors

If you try to submit the form now with no data, you will be redirected back to out /contact page and it will seem like nothing happened, but in fact, the validation was triggered and it failed because it did not pass our validation rules. required validation rule means that the field under inspection must have some data in it. If you populate all fields in our contact form correctly and submit the form you should get a blank page again.

The smart thing to do here is to provide the user some information on why the validation has failed. In the Validation documentation under [Displaying The Validation Errors](#)³² is this snippet:

Displaying validation errors

```
@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

This snippet loops through all the errors (if any) in the session and displays them in an unordered list (Twitter Bootstrap styling, but you can modify it however you want). Place this snippet just above the form tag in resources/views/contact.blade.php.

Now if you try to submit the form with no data, you should get a page which looks like this:

³⁰<https://laravel.com/docs/5.3/validation#available-validation-rules>

³¹<https://github.com/laravelista/elementary-laravel/commit/dbc88e6ccae9f8c8b0c92e9b19edb7e3cefd8949>

³²<https://laravel.com/docs/5.3/validation#quick-displaying-the-validation-errors>

Elementary Laravel
Home About Contact

Contact We value all feedback.

Feel free to contact us if you have any questions, suggestions or praises.

- The name field is required.
- The email field is required.
- The comment field is required.

Name
Your name

E-mail
Your E-mail

Message
Your message

Send

Failed Validation



Try messing around with different values to see how it works.



View changes in this commit

[62b5b9d7defe9366a97e69c1772371ddfa786350](https://github.com/laravelista/elementary-laravel/commit/62b5b9d7defe9366a97e69c1772371ddfa786350)³³.

The validation is now working. We are successfully validating the data and displaying errors to the user. We still have to process what happens if the validation is successful.

Follow the happy path

We want to redirect the user to the contact form (empty) and display a success message signaling that everything went ok.

In `resources/views/contact.blade.php`, just below our validation code place the following:

³³<https://github.com/laravelista/elementary-laravel/commit/62b5b9d7defe9366a97e69c1772371ddfa786350>

Redirecting to the contact page with a success message

```
return redirect('/contact')->with([
    'success_message' => 'Your message has been sent!'
]);
```

This code redirects the user to the /contact page. It also flashes session data with a variable called success_message. That variable will be available on our page.

Now to catch that variable on our contact page, we have to add this block of code to the place where we want it to be displayed:

Displaying the success message

```
@if (session('success_message'))
    <div class="alert alert-success">
        {{ session('success_message') }}
    </div>
@endif
```

Add this code just above the form tag in contact.blade.php. If you populate the form now with data and submit it, you will be presented with this nice little green alert box:

Elementary Laravel

Home About Contact

Contact We value all feedback.

Feel free to contact us if you have any questions, suggestions or praises.

Your message has been sent!

Name

Your name

E-mail

Your E-mail

Message

Your message

Send

Success Message



View changes in this commit

[8c4b2c8d534a7d101951fae2f04e6b2cce129fae](https://github.com/laravelista/elementary-laravel/commit/8c4b2c8d534a7d101951fae2f04e6b2cce129fae)³⁴.

Our contact form is now working, the only thing left is actually sending the email :)

Mail

Laravel provides a clean and simple API over the popular SwiftMailer library, allowing you to quickly get started sending mail.

Published at: 11. December, 2016.

Laravel 5.3 comes with a new feature called [Mailables](#)³⁵ where each type of email sent by your application is represented as a “mailable” class. In the previous tutorial we have hooked our contact form with validation and upon successful validation, presented the user with a “success” message. In this tutorial, we will simulate sending an actual email from our contact form.

Mail & Local Development



Improve your skills!

Laravel comes with drivers for many local and cloud-based services for sending emails. Check the [documentation](#)³⁶ to see how to use a specific driver.

Since we are in the development phase in our application, we don't want to actually send emails to live email addresses. To avoid doing so we will use the **Log Driver**.

Go to your local `.env` file and set a key/value for `MAIL_DRIVER=log`. Comment out all other keys that start with `MAIL_`. By doing so, all emails sent from our application will be written in the log file and not actually sent.

Mailables

Mailables are stored in `app/Mail`.

Generate a new Mailable class

To create a new Mailable enter this command:

³⁴<https://github.com/laravelista/elementary-laravel/commit/8c4b2c8d534a7d101951fae2f04e6b2cce129fae>

³⁵<https://laravel.com/docs/5.3/mail>

³⁶<https://laravel.com/docs/5.3/mail>

Creating a Mailable

```
php artisan make:mail FeedbackReceived
```

This command will create a new file `app/Mail/FeedbackReceived.php`.



There are a few things that we need to configure in our new Mailable class:

- Sender
- View
- Data

Configuring the Sender

First, we need to configure who the email is going to be “from”. We do that by setting the `from` method inside the `build` method of the `FeedbackReceived` class.

Configuring the sender

```
public function build()
{
    return $this
        ->from('you@company.com')
        ->view('emails.contact');
}
```

You can change the `from` field to anything you want or which represents your business.

Configuring the View

In the code above, we have configured the sender and specified which template should be used when rendering the email’s contents. We will now create a blank template file as specified.

Create a new folder in `resources/views` called `emails` and inside it create a file called `contact.blade.php`. Place the following code inside:

³⁷ <https://github.com/laravelista/elementary-laravel/commit/0469b561877cd6eb80667264d10480ade0792b26>

Writing Email body

```
<h1>Thank you for contacting us! Your message has been received.</h1>
```

Setting the Data

So far, we are only sending the generic confirmation message to the user who has submitted the contact form. It would be nice if we could address the user by his name and display the message that he has sent us.

To do so, we have to set public properties on our FeedbackReceived class for *name* and *comment*:

Setting class properties

```
public $name;
public $comment;

public function __construct($name, $comment)
{
    $this->name = $name;
    $this->comment = $comment;
}
```

Once the data has been set to a public property, it will be automatically available in our view as a variable. Let's modify our view template to include these variables:

Expanding the Email body with comment from Class property

```
<h1>Thank you for contacting us {{ $name }}! Your message has been received.</h1>

<p>{{ $comment }}</p>
```

Sending Mail

To send the actual email that we have configured in the previous chapter, we have to open the file `routes/web.php` and replace the TODO comment in `Route::post('contact')` with the following:

Sending Mail

```
Mail::to($request->get('email'))->send(new FeedbackReceived($request->get('name')\
), $request->get('comment')));
```

Don't forget to add the use statements above the route:

Adding use statements

```
use App\Mail\FeebackReceived;  
use Illuminate\Support\Facades\Mail;
```

Now if you populate the contact form with data that passes validation, the email will be logged in `storage/logs/laravel.log`.

Check that log file to see if the email is logged there.



View changes in this commit

[300445c0dc40d7e3099ee5602acfc98979b9a85a](https://github.com/laravelista/elementary-laravel/commit/300445c0dc40d7e3099ee5602acfc98979b9a85a)³⁸.

Congratulations! This marks the completion of the course Elementary Laravel. Thank you for reading this far.

By completing this course you will have a basic “elementary” understanding on how to use Laravel to create simple websites. The code for this course is available on Github and you are more than welcome to fork it, improve it and contribute to it.

³⁸<https://github.com/laravelista/elementary-laravel/commit/8c4b2c8d534a7d101951fae2f04e6b2cce129fae>

Laravel on Windows with Homestead

If you are getting started with Laravel and are using Windows, this is the right starting point for you. We will cover everything from installing PHP & Git to using Homestead virtual machine and creating your first blank Laravel application.

Prepare for modern PHP applications

In this introductory tutorial, we will be installing the absolute basic software that is required to run modern PHP frameworks like Laravel.

Published at: 12. March, 2016.

So, you have heard about Laravel and want to learn how to use it? Are you using Windows? Then this is the right starting point for you. In this tutorial, we will install some basic software that you will need on your PC.

These are the tools that you will be needing:

- [Git + Git Bash](#)³⁹ (2.7.0)
- [PHP](#)⁴⁰ (7.0.3 - VC14 x64 Thread Safe)

I'm running Microsoft Windows 10 x64, that is why I'm using the x64 version of PHP.
If you are on an x86 (32bit) Windows then you should use an x86 version of PHP.

Install PHP on Windows

First, we need to download PHP zip file from the website mentioned above. Unzip that file and place its content in C:\tools\php directory. Now inside that folder, you will find a file called php.ini-development. Copy/paste that file and rename it to php.ini.

There are some things that we will need to enable inside that file, so grab your favorite text editor (I prefer Sublime Text, but you can use notepad as well) and open that file:

- On line :368 change `max_execution_time = 30` to `max_execution_time = 300`. *You will thank me for this later.*

³⁹<http://www.git-scm.com/>

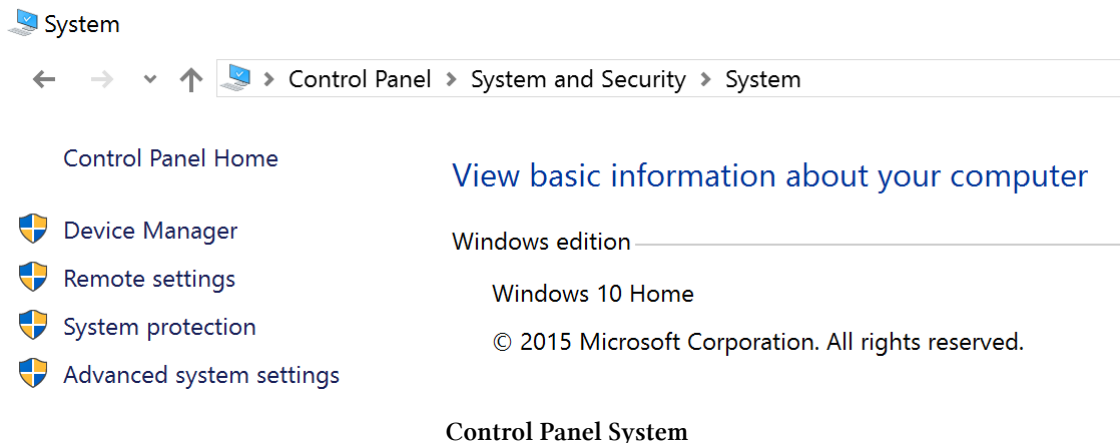
⁴⁰<http://windows.php.net/>

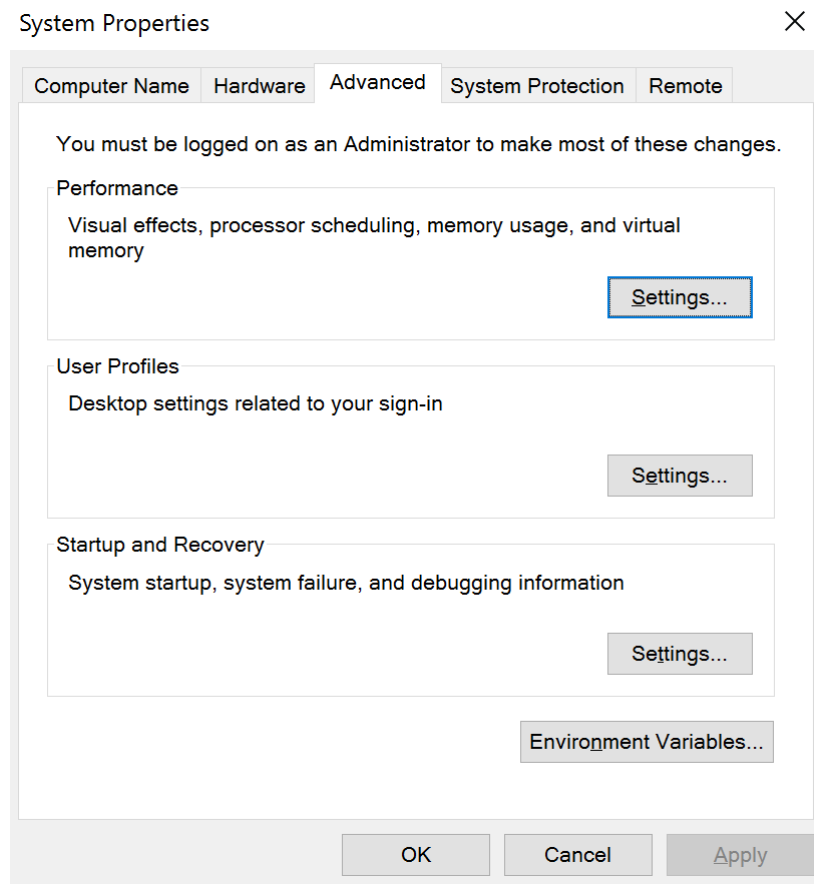
- On line :724 uncomment ; extension_dir = "ext" (Remove ; from the start of the line)
- On line :837 under section **Dynamic Extensions** you will find a list of extension. We need to uncomment a few of those:
 - extension=php_curl.dll
 - extension=php_fileinfo.dll
 - extension=php_gd2.dll
 - extension=php_mbstring.dll
 - extension=php_mysqli.dll
 - extension=php_openssl.dll
 - extension=php_pdo_mysql.dll
 - extension=php_pdo_sqlite.dll

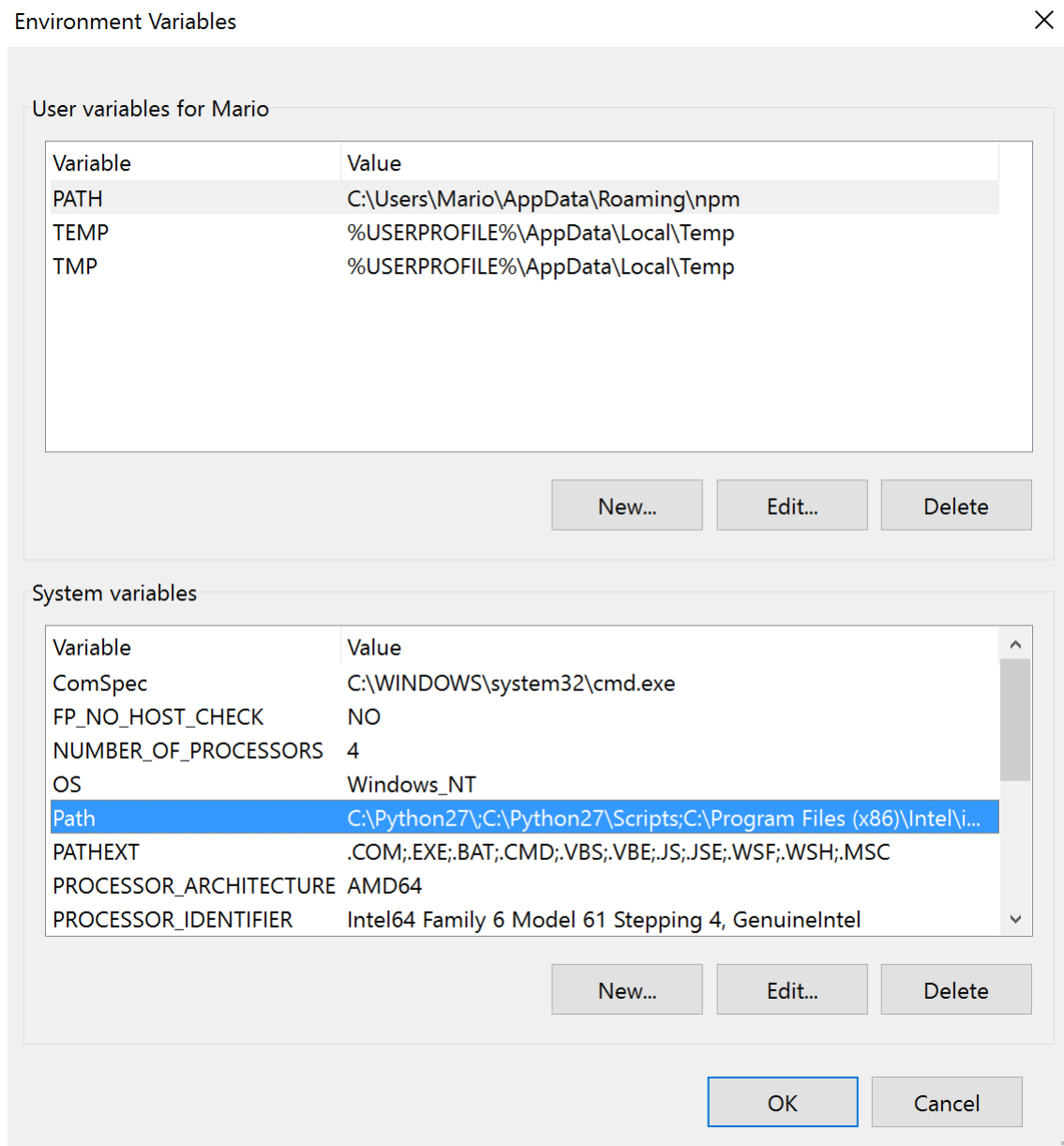
Now we have configured PHP for Laravel and other modern PHP applications.

We still have to tell Windows where to find this PHP installation. And we do so by adding the absolute path of the folder we installed PHP to our System Path Environment Variable.

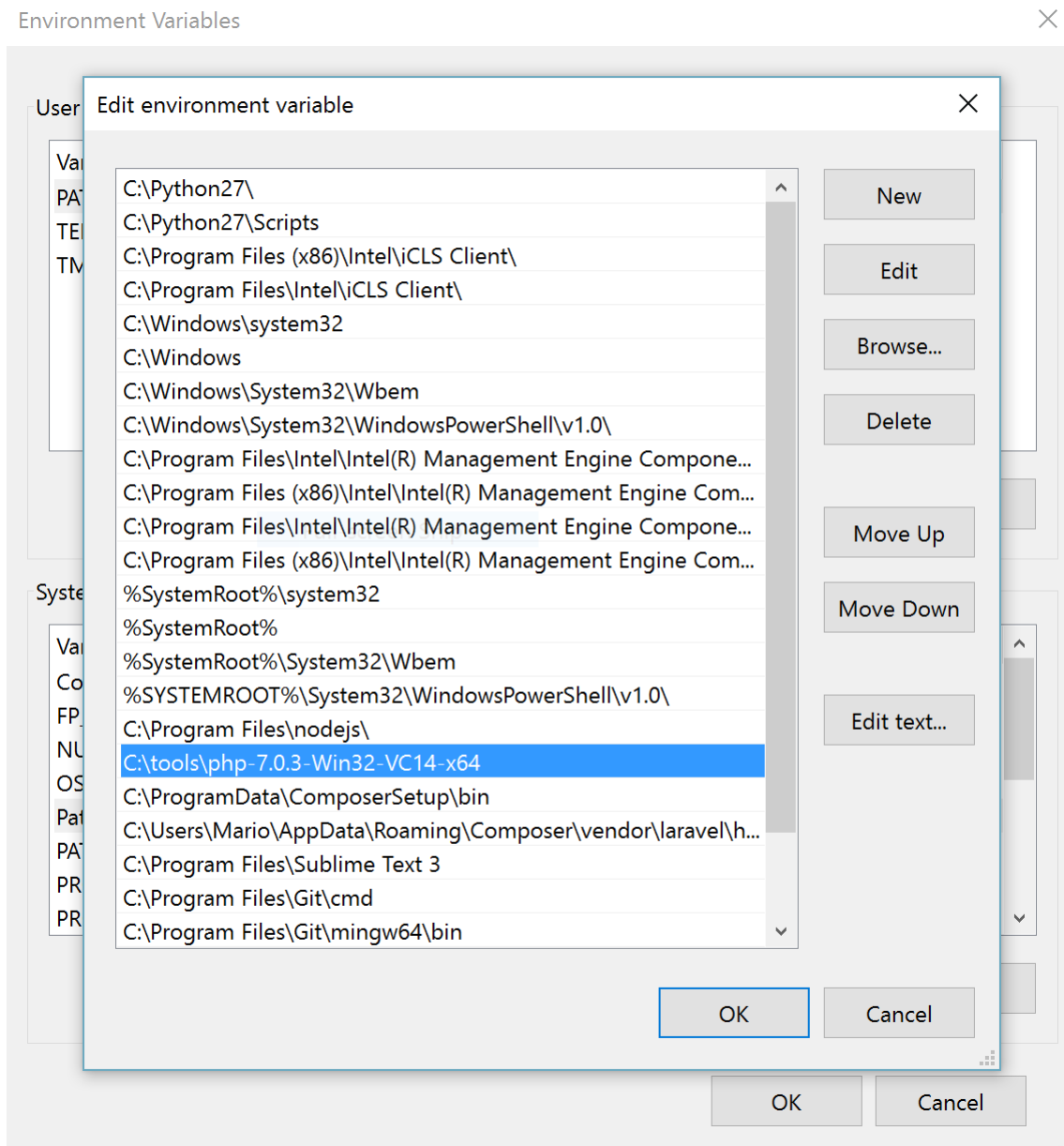
Open Control Panel and go to Control Panel\System and Security\System\Advanced system settings\Advanced\Environment Variables and under System variables locate Path press *Edit*. Here add a new value pointing to your PHP installation folder that contains php.ini file. In my case, I would add C:\tools\php-7.0.3-Win32-VC14-x64. Press *Ok* to all and close open windows.



**Advanced system properties**



Environment variables

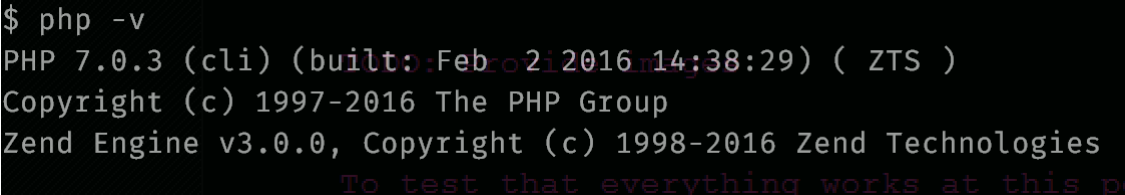


Path system variables

To test that everything works at this point. Open Command Prompt `cmd` and type `php -v`. You should get something like this:

Checking PHP version

```
$ php -v
PHP 7.0.3 (cli) (built: Feb  2 2016 14:38:29) ( ZTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
```



```
$ php -v
PHP 7.0.3 (cli) (built: Feb  2 2016 14:38:29) ( ZTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
To test that everything works at this point
```

php version output

Great, now we can move on. If you are having problems at this point, leave me a comment below and I will help you out.

Install Composer

Go to the [download page for Composer](#)⁴¹ and download the Windows installer. It will install the latest version and configure everything on your system.

You can verify that everything is working by typing `composer --version` in the terminal.

Checking Composer version

```
$ composer --version
Composer version 1.0-dev (72cd6afdfce16f36a9fd786bc1b2f32b851e764f) 2015-12-28 1\
7:35:19
```

Install Git and Git Bash

If you are building a modern PHP application or planning to use Laravel you should really learn how to use Git, because without it, over time you will get yourself in a big mess.

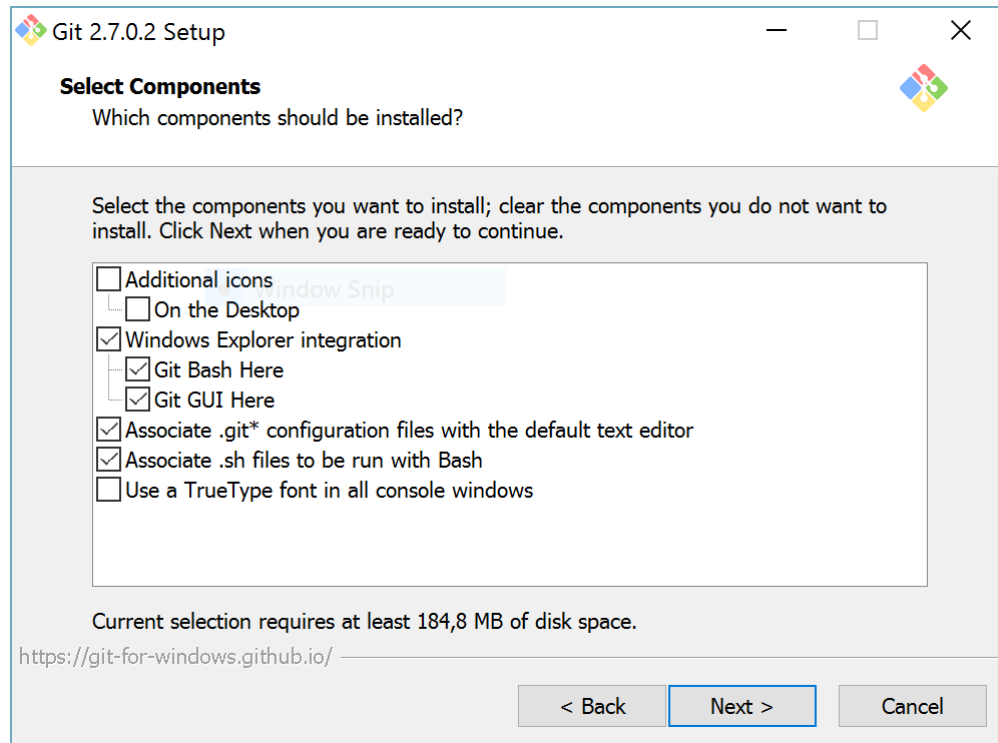
The installation is pretty simple, just download the [latest version of Git](#)⁴² and complete the installation with these options:

- Use MinTTY (the default terminal of MSys2)
- Checkout WIndows-style, commit Unix-style line endings

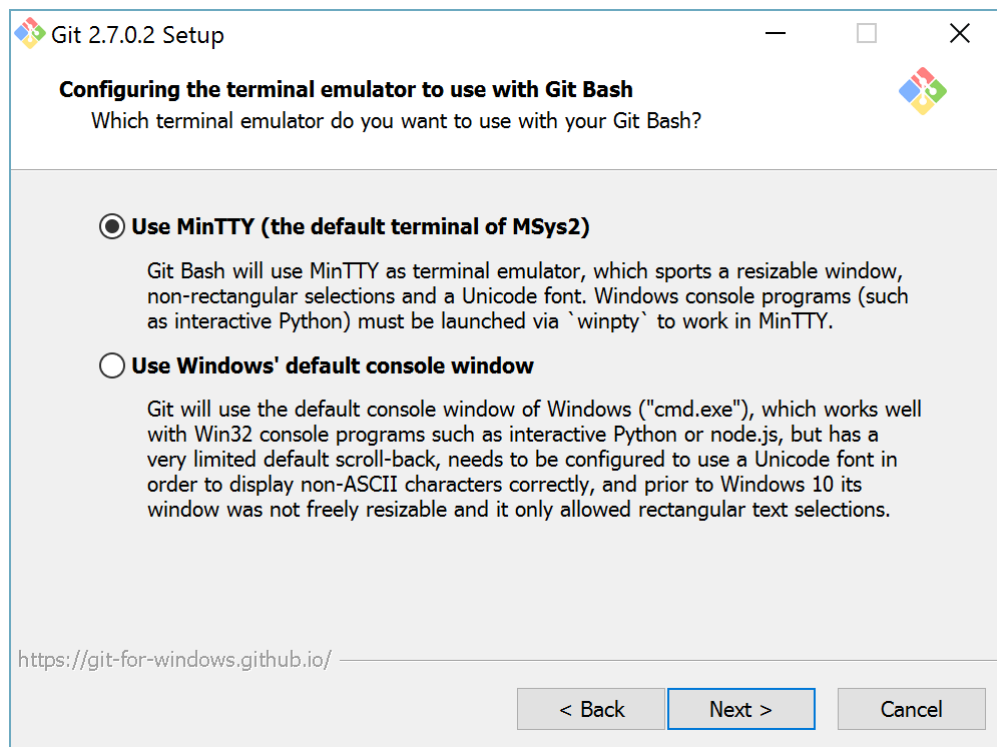
⁴¹<https://getcomposer.org/download/>

⁴²<http://www.git-scm.com/download/win>

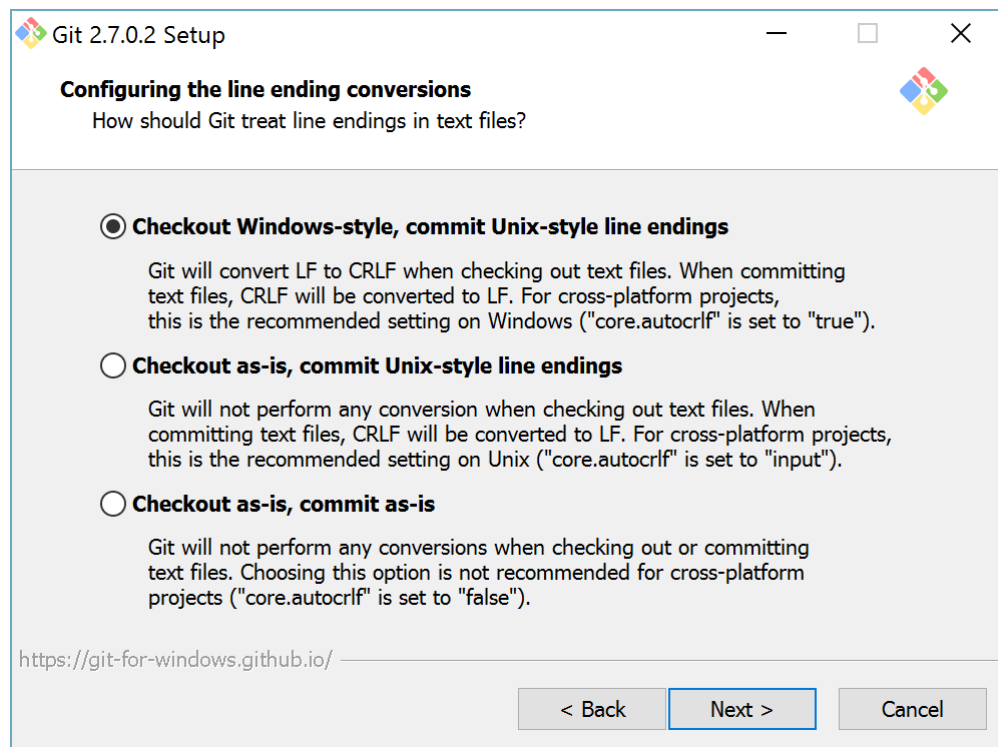
- Use Git and optional Unix tools from the Windows Command Prompt
- Enable file system caching



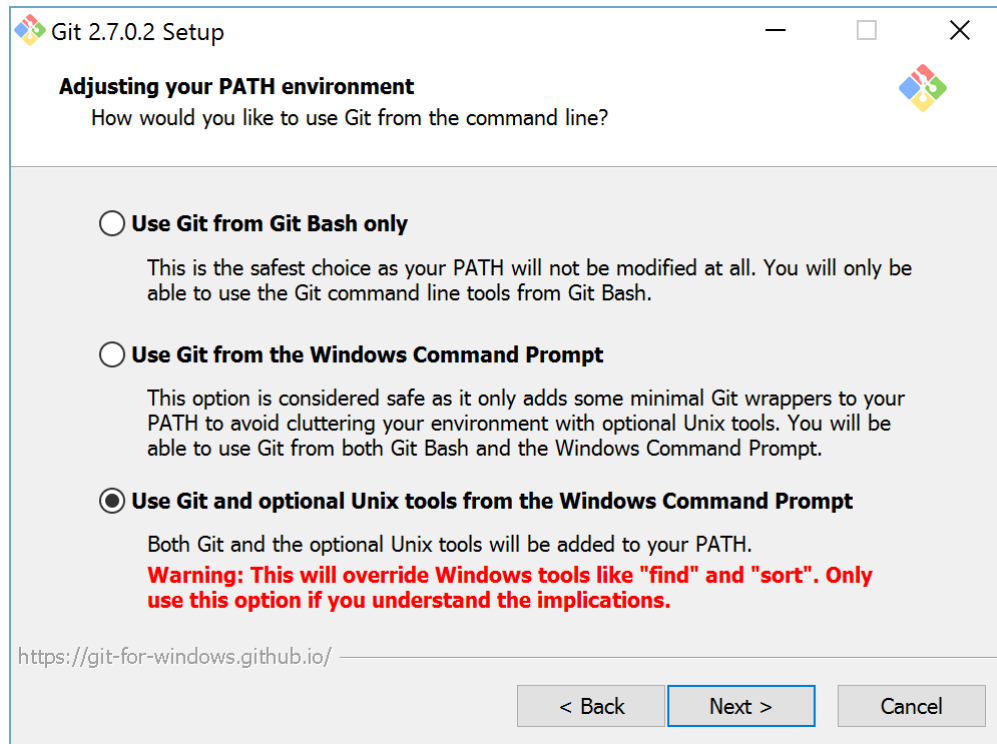
Git Components



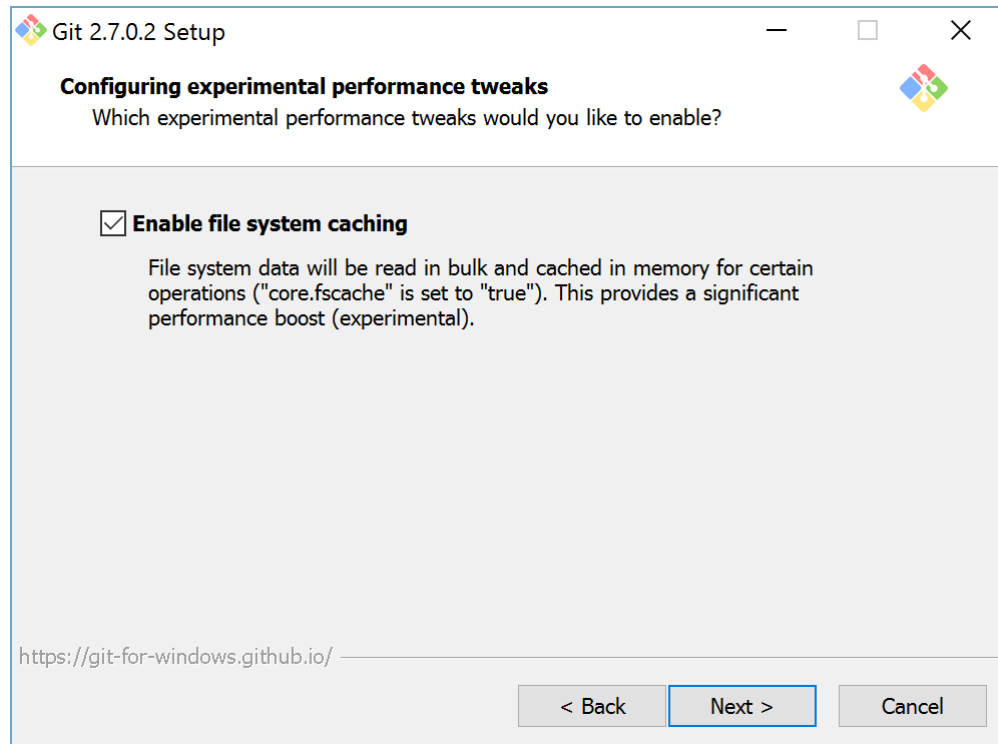
Git Terminal Emulator



Git Line ending Checkout style



Git Environment variables



Git File System Caching

To test that everything is working, find **Git Bash** under Programs, right-click on it and click **Run as Administrator**. Type `git --version` and you should get:

Checking Git version

```
$ git --version
git version 2.7.0.windows.2
```

You can customize the appearance of the terminal by going to options and changing the font family, font size, transparency, full screen and many other options.

I'm using Fira Code, 11pt, medium transparency, scrollbar turned off and xterm-256color terminal.

Install Node.js

You will be needing node.js to install NPM modules and use [Elixir](https://laravel.com/docs/5.2/elixir)⁴³.

Go to [node.js website](https://nodejs.org/en/)⁴⁴ and download the latest stable version (v5.5.0). The installation is pretty straight forward, just follow the installer. You can verify your installation by typing `node -v`.

⁴³<https://laravel.com/docs/5.2/elixir>

⁴⁴<https://nodejs.org/en/>

Repositories location & text editor

Because of path length limitation on Windows, I suggest that you place all of your repositories in the root of your drive C:\repositories. *This solves many issues with npm*

For you text editor or IDE I suggest using [Sublime Text 3](#)⁴⁵ or [PHPStorm](#)⁴⁶. However, you are free to use anything you want.

You are now ready to move on to the next tutorial.



Important!

When I say “use the terminal” or “type in terminal” in future tutorials, that means to use **Git Bash** program we installed in this tutorial. Everything you do from this point on in terminal, you should be done in **Git Bash** console.

Getting started with Homestead

Your own local virtual server for running PHP applications with lots of extra software in case your projects requires it.

Published at: 12. March, 2016.

What is Homestead and why all the fuss about it?

Taken from [official Laravel documentation](#)⁴⁷ on Homestead:

Laravel Homestead is an official, pre-packaged Vagrant box that provides you a wonderful development environment without requiring you to install PHP, HHVM, a web server, and any other server software on your local machine.

In the simplest way; everything you need (development server related) to start working on your Laravel application is already included in Homestead. You just need to set it up and you are good to go.

Virtualbox provides you with an ability to manage virtual machines. Vagrant is used for automating the virtual machine creation process. Homestead is a Vagrant box. You tell Vagrant to use the Homestead box to create a virtual machine using Virtualbox and voila everything is up and running.

⁴⁵<http://www.sublimetext.com/3>

⁴⁶<https://www.jetbrains.com/phpstorm/>

⁴⁷<https://laravel.com/docs/5.2/homestead#introduction>

Let's start from the start :)



Important!

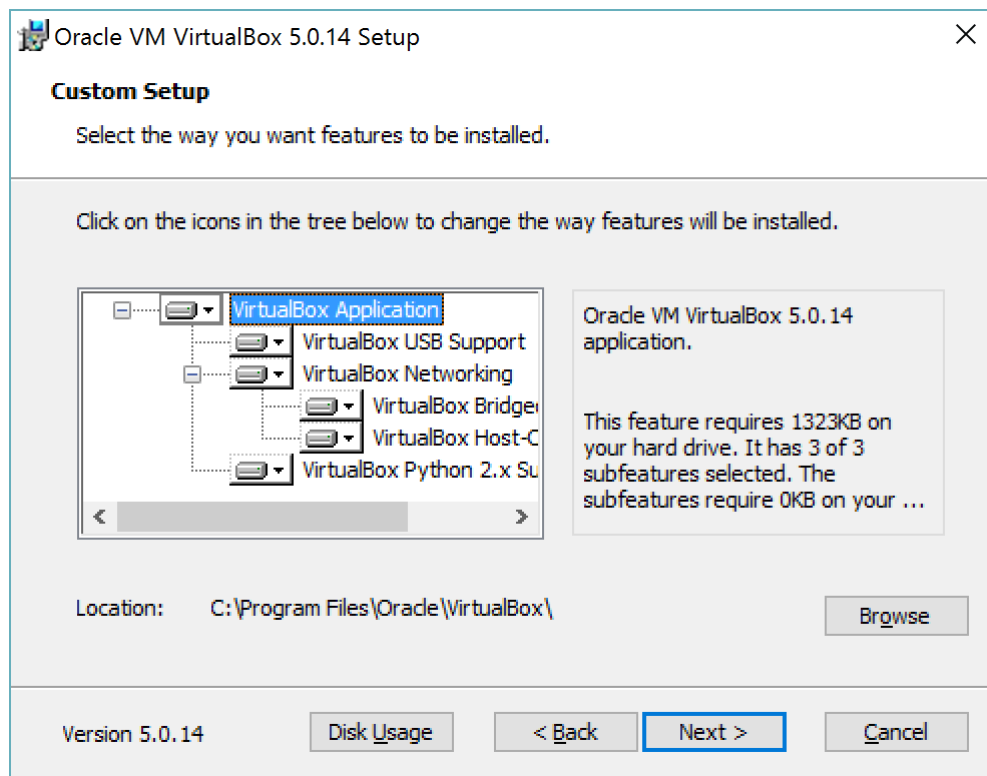
When I say "use the terminal" or "type in terminal", that means to use **Git Bash** program we installed in the previous tutorial. Everything you do from this point on in terminal, you should be doing in **Git Bash** console.

Install Virtualbox

Visit the official Virtualbox download page and download the latest version (At the time of writing this tutorial the latest version is 5.0.14). Once downloaded run the setup.

For reference I'm running Microsoft Windows 10 x64.

Leave all the defaults on this step.



Features

And complete the setup by pressing next to everything as usual :) This will disable your network connection for a few seconds so keep that in mind if you are doing something online like reading this tutorial.

Also, be sure to download and install VirtualBox Extension Pack from the same download page. *The extension Pack version must match Virtualbox version.*

Now that we have installed Virtualbox and the Extension Pack we will proceed to Vagrant, but before we do, be sure to restart your PC.

Install Vagrant

Go to the [Vagrant download page](#)⁴⁸ and download the latest version (1.8.1).

Once downloaded, complete the setup by pressing next to everything.

To test that everything is working run `vagrant -v` from the terminal.

Checking Vagrant version

```
$ vagrant -v
Vagrant 1.8.1
```

Install The Homestead Vagrant Box

Run the following command in the terminal to download the latest Homestead box:

Adding Homestead vagrant box

```
vagrant box add laravel/homestead
```

This command should take some time depending on your download speed.

You now have Vagrant installed.

Install Homestead

To install Homestead clone the repository in your Home (~/) directory with the following command:

Cloning Homestead repository

```
cd ~

git clone https://github.com/laravel/homestead.git
```

Navigate to that directory and run `bash init.sh` to create necessary files. You should get an output similar to this:

⁴⁸<https://www.vagrantup.com/downloads.html>

Initializing Homestead

```
$ ./init.sh
```

```
Homestead initialized!
```

This means that the `Homestead.yaml` file has been placed in the `~/homestead` hidden directory along with two other files (You can open those files with any text editor to see what they are for).

Now open `Homestead.yaml`. This file is the file in which you will be making any future changes. You can:

- change virtual machine settings
- set your SSH key
- add folders
- add Nginx sites
- add databases and more...

For start, change `folders` to point to the directory where you keep your repositories, like so:

Setting the path to your repositories folder

```
folders:  
  - map: C:/repositories  
    to: /home/vagrant/repositories
```

There will be more talk about this file later when we will create a blank Laravel application.

Daily usage

In order to avoid navigating to `C:\repositories\homestead` directory every time you want to start the virtual machine, you can add a simple Bash alias to your Bash profile.

Go to `~/` and check if you have a hidden file there called `.bash_profile`. If you don't have that file create it and place the following inside it:

Adding Bash aliases and functions

```
# Some shortcuts for easier navigation & access
alias ..="cd .."
alias vm="ssh vagrant@127.0.0.1 -p 2222"

# Homestead shortcut
function homestead() {
    ( cd /c/repositories/homestead && vagrant $* )
}
```

Save the file and restart your terminal for the changes to take effect.

Now is the time to finally start Homestead. Now, you must open Git Bash (terminal) as an administrator (Right click -> run as administrator). Start homestead by typing homestead up in your terminal. Vagrant will boot the virtual machine and automatically configure your shared folders and Nginx sites.

Your first Laravel application

Learn how to create a blank Laravel application and serve it locally with Homestead on your PC for development purposes.

Published at: 12. March, 2016.

To sum things up, so far we have installed the necessary software on our host PC, Virtualbox, Vagrant and got Homestead up and running. Now we will create a blank Laravel application, configure Homestead to serve it and change the hosts file so that we have a custom domain for our application.

This process you will have to do every time you create a new Laravel application so try to remember it.

Repositories root directory

As mentioned in the previous tutorial you need to have a repositories folder where you keep all your repositories/applications. In .homestead/Homestead.yaml you have a line:

Repositories folder location

folders:

```
- map: C:/repositories
  to: /home/vagrant/repositories
```

Navigate to that folder and follow the instructions below.

Install Laravel via installer

First, we have to download the Laravel installer by typing the command below in terminal:

Installing Laravel installer

```
composer global require "laravel/installer"
```

This command will download the Laravel installer and create an executable that you can call upon.

To be able to use laravel installer from the terminal we first have to add composer bin directory to our path. We do that by going to Control Panel\System and Security\System\Advanced system settings\Advanced\Environment Variables and under System variables locate Path press *Edit*. Here add a new value pointing to the composer bin directory. *For reference mine is C:\Users\Mario\AppData\Roaming\Composer\vendor\bin.*

To create a directory containing a fresh Laravel installation with all dependencies installed use this command in your repositories root directory:

Creating a new Laravel project

```
laravel new myblog
```

This method of installation is much faster than installing via Composer.

Now you have a directory called myblog and inside it, your first Laravel application. **Here are a few tips when working on Windows.** Navigate to your app and open it using Sublime Text like so:

Opening the folder using Sublime Text

```
cd myblog
```

```
subl .
```



If your system can't find `subl` you need to add it to your path. The procedure is the same as the above. Add `C:\Program Files\Sublime Text 3` to your Path.

Now open the file called `.env`. This file contains all configuration options for your application and by default is not included in version control.

- change `DB_HOST=127.0.0.1` to your Homestead machine IP address. By default that is `192.168.10.10`. This will enable you to run migrations and tinker with your application from the host PC without the need to ssh into Homestead.
- change `DB_DATABASE=homestead` to something more meaningful like `myblog`.

That's it! Now we have to tell Homestead about our brand new application.

Add application to Homestead

To add a new Nginx site to Homestead we need to open `~/homestead/Homestead.yaml` and add a new site for `myblog.app` and create a database `myblog`:

Adding application to Homestead

```
sites:
  - map: myblog.app
    to: /home/vagrant/repositories/myblog/public

databases:
  - myblog
```

Save the changes and type `homestead provision` in the terminal (remember you must run Git Bash as an Administrator every time your work with Homestead). This command will preserve all changes: sites, databases, custom modifications and update it with new sites and databases. **Very useful.**

There is one more step before you can access your brand new blog and that is adding our custom domain `myblog.app` to point to our Homestead machine IP `192.168.10.10`.

The Hosts file

The hosts file will redirect requests for your Homestead sites into your Homestead machine. On Windows, it is located at `C:\Windows\System32\drivers\etc\hosts`. The lines you add to this file will look like the following:

Updating hosts file

```
192.168.10.10  myblog.app
```

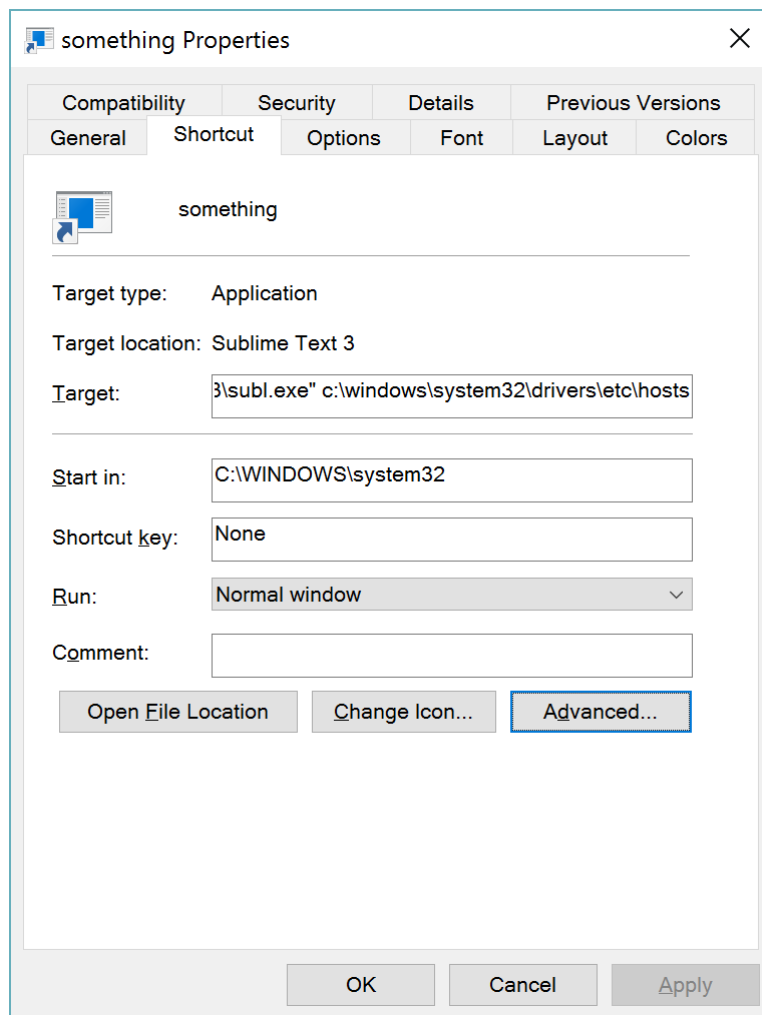
There are a few glitches here. In order to save changes to the hosts file, you must open it as an administrator. That means opening the terminal as an administrator navigating to `C:\Windows\System32\drivers\etc` and opening the file hosts using Sublime Text. *You will be doing that a lot.*

So to save you and myself time, I have created a shortcut for that.

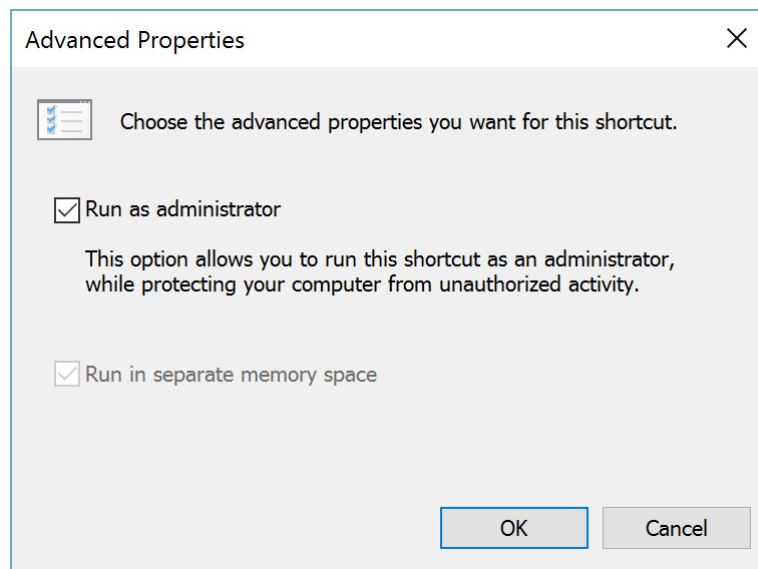
The shortcut

Go to Desktop and create a new shortcut (right click -> new -> shortcut). When asked for the location of the item paste "`C:\Program Files\Sublime Text 3\subl.exe`" `c:\windows\system32\drivers\etc\hosts` and press Next. Now type the name for this shortcut `Edit Hosts` and press Finish.

Now right click on the shortcut and go to Properties -> Shortcut -> Advanced and check the box saying Run as Administrator. This opens the hosts file as an administrator so that you can save changes.



Advanced properties



Run as Administrator

Cut that shortcut (Ctrl+X or right click -> cut) and using the File Explorer navigate to C:\ProgramData\Microsoft\Windows\Menu\Programs and paste it there. It will require you to confirm that you are an administrator.

Once you have done all of this, it is easy now to edit the hosts file. Press the Windows key on your keyboard and start typing Edit Hosts. The shortcut that we have just created will show. Press Enter and add this line at the bottom:

Updating hosts file

```
192.168.10.10 myblog.app
```

Save the file and you are done.



Sublime Text must be closed before you run this shortcut. Otherwise, if Sublime is opened in normal mode you cannot save changes to the hosts file.

Once you have added the domain to your hosts file, you can access the site via your web browser:

Accessing the website in the browser

```
http://myblog.app
```



Laravel 5

Laravel

Now you have your first Laravel application running on Homestead. This tutorial concludes the course Laravel on Windows.

Laravel on Windows with Laragon

Another approach to getting started with Laravel on Windows is by using Laragon. Laragon offers you a fast, powerful and Isolated Development Environment. It is portable, very flexible and doesn't affect your operating system.

Hello Laragon

In this tutorial, I will tell you about Laragon which is an alternative to Homestead and we will also cover the entire process of installation.

Published at: 04. April, 2016.

So, what is Laragon and you should use it over Homestead on Windows. In my previous course called [Laravel on Windows](#) I've got a few comments saying that I should mention Laragon as an alternative to Homestead for users that are unable to perform the steps in the tutorial (by [landjea](#)⁴⁹) and because it is less hassle to set up things natively and having to worry about crap associated with VM's (by [Matthew Rath](#)⁵⁰).

I've taken those comments into consideration, took some time to explore Laragon and have come up with this course where I will explain what Laragon is when you should use it and what are its features.

As stated on the [official website](#)⁵¹, Laragon is a fast, powerful and Isolated Development Environment. It is portable and very flexible.

Installing Laragon is effortless & doesn't affect your OS (Windows). You can move Laragon folder around (to another disk, to another laptop, sync to Cloud,...) and it still works.

To even more simplify this, Laragon is a WAMP (Windows, Apache, MySQL, PHP); Windows web development environment. It does not affect your operating system. You install it as a software, start it up, do your programming and when finished you just exit.

When to use Laragon over Homestead?

⁴⁹<https://disqus.com/by/landjea/>

⁵⁰https://disqus.com/by/matthew_rath/

⁵¹<https://laragon.org>

This is difficult to answer because it depends on many factors. Since Homestead is the officially supported way of running Laravel I would recommend using it, but if for some reason you can't (don't have administrator rights or unable to run a VM) the next best thing is Laragon.



You can always use `php artisan serve` and SQLite database to avoid using Homestead or Laragon if you wish. *This assumes that you have PHP installed on your OS.*

This introduction is long enough, let's move on to the fun stuff.

Features

One thing that I find lacking on the Laragon website is the summary of current features. On the official website, you have the download link and the link to the forum. No install instructions or anything similar can be found on the front page.

It took me some time, to summarize all the features from the [Announcements category](https://forum.laragon.org/category/1/announcements)⁵² on the Forum and even more time to find out how to use some features.

Don't worry, as it turns out it is all very simple.

Software and services that you get with Laragon 1.0.7 are:

- [Cmder](http://cmder.net/)⁵³
- Git
- Node.js
- NPM
- SSH
- Putty
- PHP 7 & 5.6 (Easily switchable with one click)
- Activate/deactivate PHP extensions on the fly
- xDebug
- Composer
- Apache
- MariaDB/MySQL
- phpMyAdmin
- Full Lumen and Laravel support
- Auto create virtual hosts

⁵²<https://forum.laragon.org/category/1/announcements>

⁵³<http://cmder.net/>

- Mail Catcher - Laragon will show a small window on the bottom right of your screen and help you quickly view content of the generated email
- Mail Sender - You can use `mail()` function to send mail to the Internet easily and effortlessly
- Mail Analyzer: Analyze what happens when an email is sent and show helpful information to make sure that your email configurations are correct.
- ngrok - allows connections from the Internet to the local server

Useful shortcuts

Global hotkey to open shell (cmd): CTRL+ALT+T

Shell shortcuts:

Useful shortcuts

```
e -> open notepad++  
e. -> open explorer  
ll -> list current dir with full information  
vi -> if you love vim
```

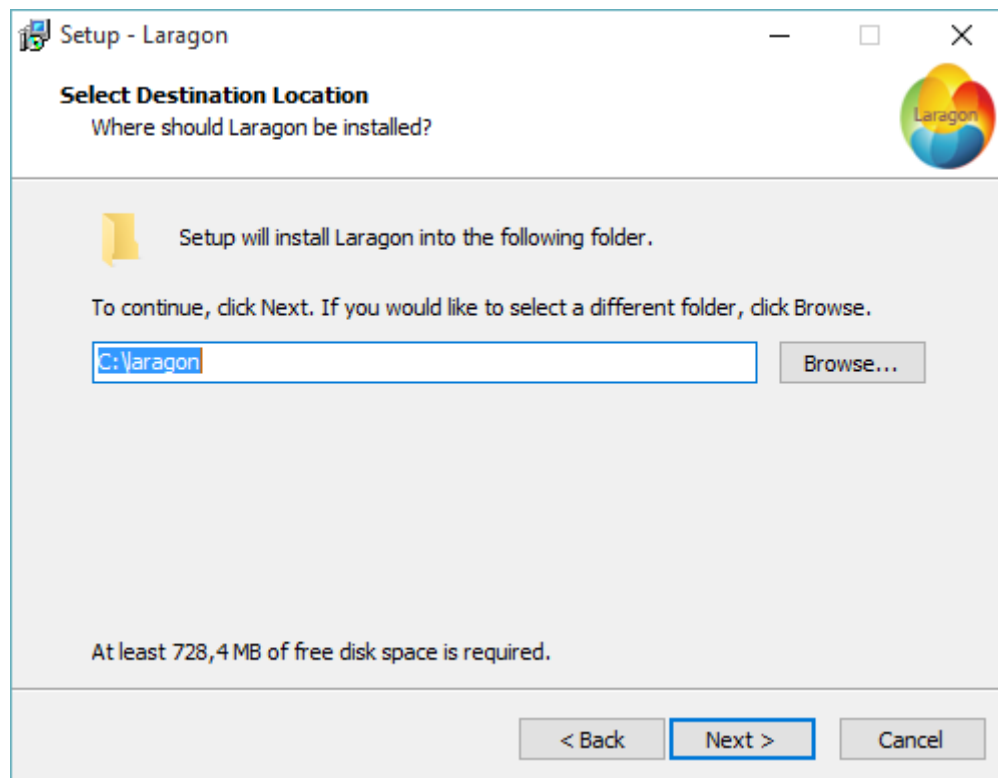
Now that we know what Laragon is and what are its features we can move on to installation.

Installation

The installation is very simple, just click the download button on the [official website](https://laragon.org)⁵⁴ and follow the installer.

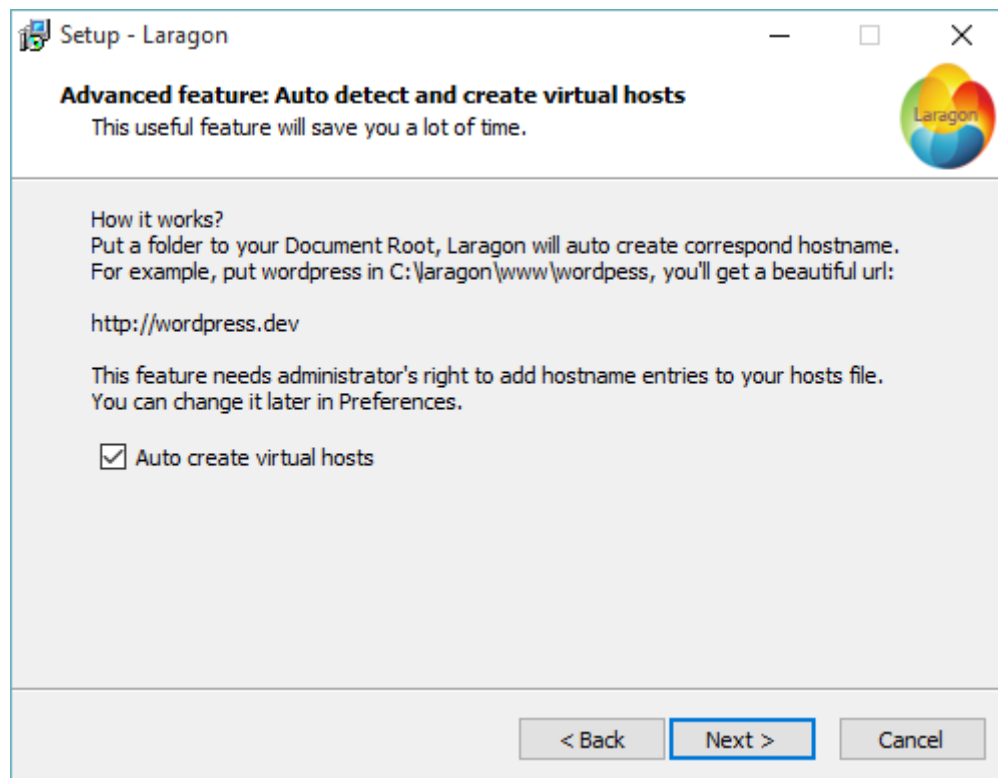
You can choose where to put the Laragon folder (Later you can move this folder where ever you want, but I suggest placing it in the root of any drive):

⁵⁴<https://laragon.org>



Installation Directory

Be sure to enable *Auto create virtual hosts* feature:



Auto enable virtual hosts

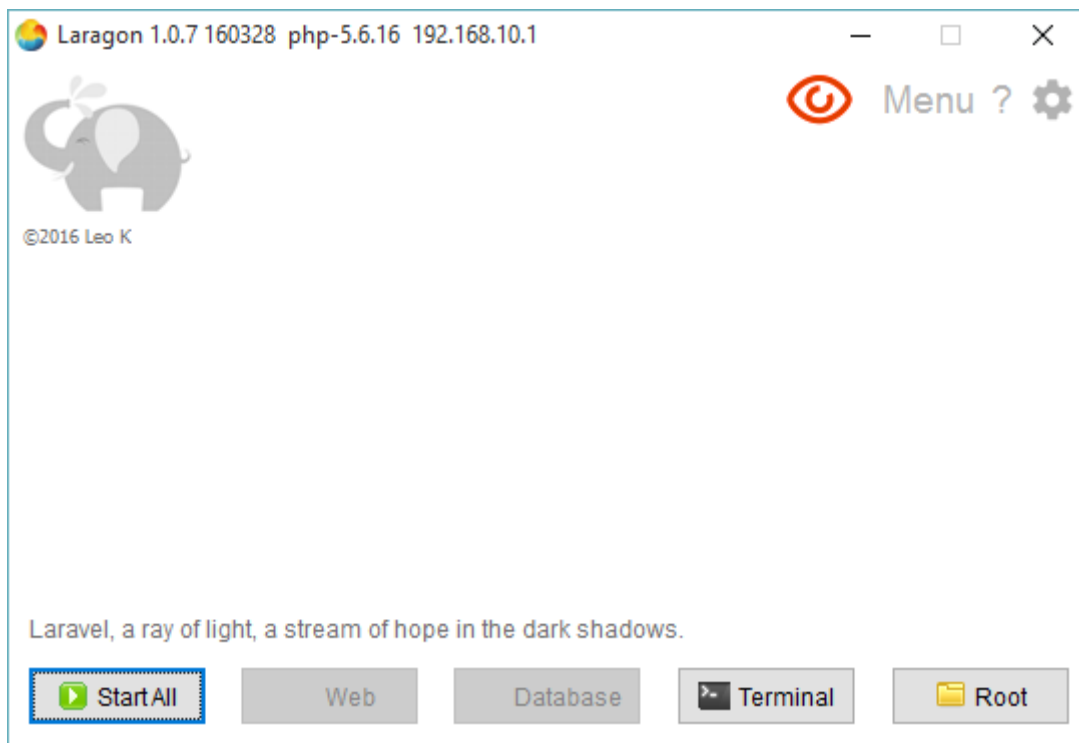
Great, now the installation is complete, but don't start Laragon yet.

Auto create virtual hosts

Once the installation has completed, you have to decide if you want to use the *Auto create virtual hosts* feature or not. If you want to use it, you must **run Laragon as an administrator**. If not, you can run it as a normal user but then that feature will not work.

This feature converts project folder name in `C:\laragon\www\` to a friendly domain name. If your projects folder is called *superawesomewebsite* then Laragon will create a local domain which you can access at `http://superawesomewebsite.dev`

Now let's run Laragon as an administrator. You should see a screen like this:

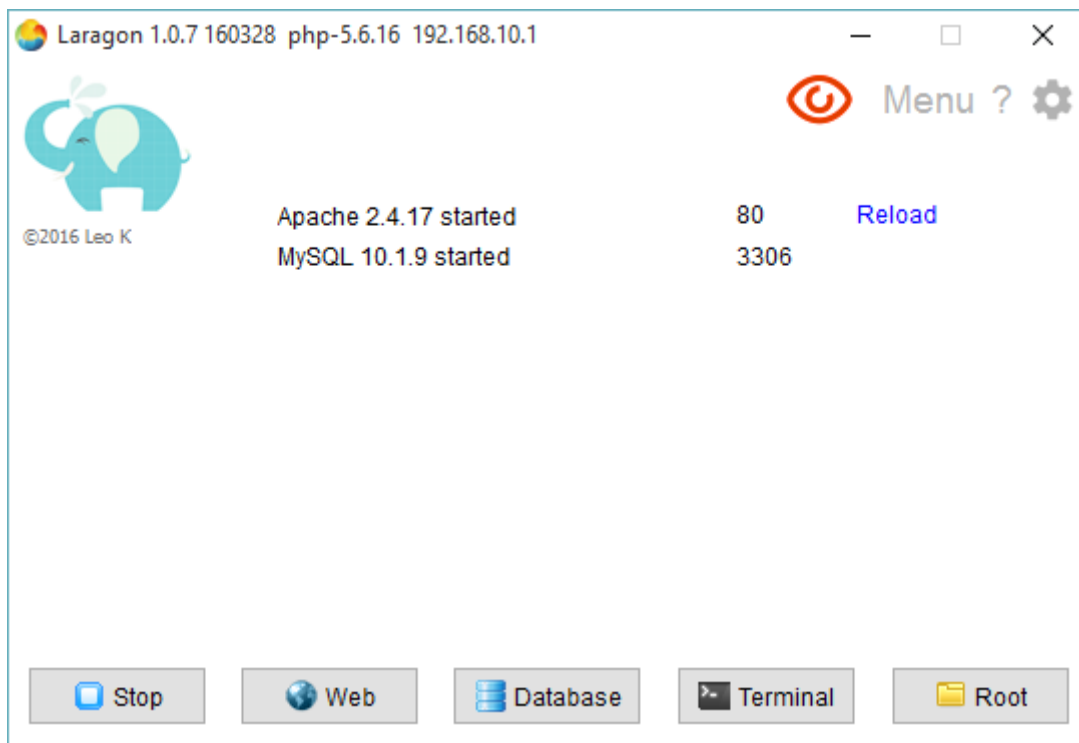


Start screen

Switch PHP versions

Before we click on *Start All* button, you can decide which version of PHP you want to be used. Go to Menu -> PHP -> Version and choose the one you want. I prefer to use PHP 7 :)

Now click on *Start All* and you should get Apache and MySQL running.



All services started

You now have Laragon installed and running.

Your first Laravel application

Learn how to create a blank Laravel application and serve it locally with Laragon on your PC for development purposes.

Published at: 04. April, 2016.

In the previous tutorial, we have installed Laragon and started its services. *I am running Laragon as an administrator so that the *auto create virtual hosts feature is enabled**. In this tutorial, I will show you two ways on how to start a brand new Laravel 5 application.

First, using the GUI of Laragon and then using the shell (cmd).

Repositories root directory

To find out where Laragon stores its projects click on *Root* button on the GUI (Graphical User Interface). It will open File Explorer on repositories root directory. You can see the full path in

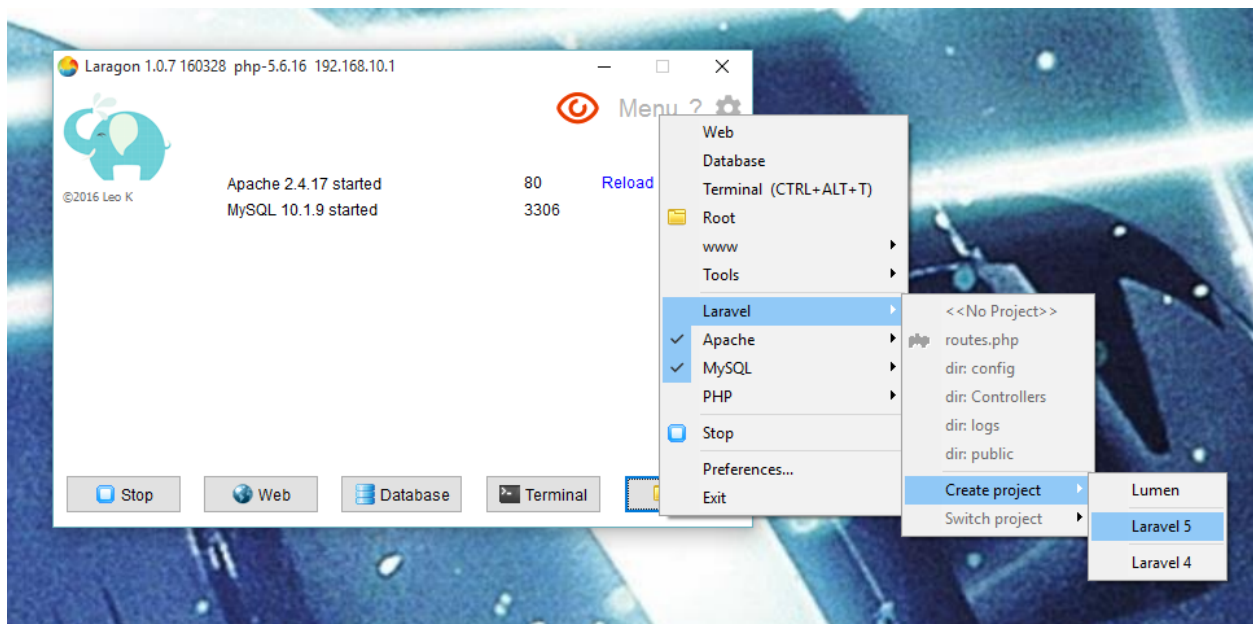
the path bar above: `C:\laragon\www`; if you left the default installation folder of Laragon during the installation.



This is the location where you should place your existing projects if you have any yet.

Install Laravel via GUI

Now, this is the easiest way of installing Laravel 4, 5 or Lumen I have ever seen. Click on Menu -> Laravel -> Create project -> Laravel 5.



Install Laravel 5

You will be asked for the project name. Enter the name and confirm. A command line window will open and the installation of Laravel will begin.

I have entered `laravel5` as project name, but you can choose how you want to name your projects. In future reference, you can replace `laravel5` with the project name you entered.

Once completed you will see a message:

Laragon message

Run Laragon as Administrator to get beautiful URL:
`http://laravel5.dev`"

Before you visit that URL, be sure to press *reload* on Laragon GUI for changes to take effect. If you visit the URL `http://laravel5.dev` in your browser you will see a welcome screen of Laravel 5.



Laravel 5

Laravel 5 Welcome

That's it! Your project is now located in `C:\laragon\www\laravel5` folder. You can use any text editor or IDE to open it and start working on your brand new Laravel 5 application.

Install Laravel via shell

If for some reason you don't want to use the GUI to create a Laravel 5 project, you can always use the shell aka Terminal.

On the Laragon GUI click on *Terminal* to open the Cmder shell which points to your repositories root directory. Or there is a global keyboard shortcut mentioned in the previous tutorial `CTRL+ALT+T` which opens the same Cmder shell.

In shell type:

Installing Laravel project using Composer

```
composer create-project laravel/laravel your-project-name --prefer-dist
```

to create a new Laravel project. *This is mentioned in the official Laravel documentation under [Installation](#)⁵⁵.*

Now you have your first Laravel application running on Laragon.

Remote access using ngrok

Enable your friends and clients to access your application over the Internet, while working on it locally.

Published at: 18. April, 2016.

Let's dive right into this tutorial. *What do we want to do?*

We want to expose our local application to the Internet so that our clients or friends can view what we are working on or the current progress of the project while it is still not on production server.

Secure tunnels to localhost - [ngrok](#)^a

^a<https://ngrok.com/>

Some of the benefits of using ngrok from their website:

- Demo without deploying
- Simplify mobile device testing
- Build webhook integrations with ease
- Run personal cloud services from your own private network

This is a really nice feature that Laragon has out-of-the-box.

Remote Access

Let's continue where we left off in the previous tutorial. We have created a blank Laravel application on <http://laravel5.dev> domain and all Laragon services are running.

On the Laragon GUI click on *Terminal* to open the Cmder shell **Or** use the shortcut CTRL+ALT+T which opens the same Cmder shell.

In shell type:

⁵⁵<https://laravel.com/docs/5.2/installation>

Enabling remote access using ngrok

```
ngrok http laravel5.dev:80
```

You will get an output similar to this one:

```
ngrok by @inconshtreveable

Tunnel Status      online
Version            2.0.25/2.0.25
Region             United States (us)
Web Interface      http://127.0.0.1:4040
Forwarding          http://1a1aed8f.ngrok.io -> laravel5.dev:80
Forwarding          https://1a1aed8f.ngrok.io -> laravel5.dev:80

Connections
  ttl    opn    rt1    rt5    p50    p90
    1      0     0.00   0.00   8.94   8.94

HTTP Requests
-----
GET /favicon.ico    404 Not Found
GET /               200 OK
```

ngrok output

Take a note of this line:

Obtaining remote access URL

```
Forwarding http://1a1aed8f.ngrok.io -> laravel5.dev:80
```

In my case this is `http://1a1aed8f.ngrok.io`, but your output will be different, so keep that in mind and write it down somewhere. You will need it for the next step.

Now go to the Laragon GUI and click on Menu -> Apache -> `http-vhosts.conf` and add the domain from ngrok to `ServerAlias` of `laravel5.dev` virtualhost entry.



Keep in mind that if you have named your project differently than *laravel5*, use your project name instead.

Now my virtual host entry looks like this:

Virtual host entry for Apache

```
<VirtualHost *:80> #laragon magic!
    DocumentRoot "C:/laragon/www/laravel5/public/"
    ServerName laravel5.dev
    ServerAlias *.laravel5.dev 1a1aed8f.ngrok.io
</VirtualHost>
```

One last thing before this starts working is to restart Apache. You do that by going to the Laragon GUI and clicking on Menu -> Apache -> Reload or just click on Reload on the GUI screen next to the Apache service.

You can now access your local application over the Internet using the given URL from ngrok. In my case, this is <http://1a1aed8f.ngrok.io>.



Laravel 5

ngrok remote



The URL will be changed each time you run ngrok.

Quickstart

1. Run ngrok for project ngrok `http project.dev:80`.

2. Add URL given from ngrok to ServerAlias of that project in Apache `http-vhosts.conf` file (Menu -> Apache -> `http-vhosts.conf`).
3. Reload Apache (Menu -> Apache -> Reload).
4. Send given ngrok URL to friend, client etc...

This was a nice and simple tutorial on a thing that is usually very complicated to do, but thanks to Laragon it was a breeze.

Configure Laravel 5 for Shared Hosting

There are a few things that you have to change in Laravel to make it work on shared hosting. The most important thing is to change the public path and correctly bootstrap the application.

Published at: 29. July, 2016.



View Source Code

Source code for this tutorial is available [here](#)⁵⁶.

I recently had to build a website that would be used on shared hosting. At first, I went with a standard HTML `index.html` file and then create a file called `contact.php` to handle the contact form submit action. The website can be found at studio-renata.hr⁵⁷ as soon as they move the website to the server that has PHP 5.6 version.

As I started coding the website the number of lines started to grow and I had a lot of duplicating syntax for stuff like images, containers etc.. Somehow I got over it and was feeling happy with it, but then I had to deal with validating the contact form using AJAX. As you can guess I did a quick research and found out that Laravel can run on shared hosting, but with a few limitations:

- There is no console (artisan)
- You cannot use Composer to install/update
- You cannot use Git to version your application

That being said, there are a few requirements that your server needs to have in order for Laravel to work:

- PHP \geq 5.5.9
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension
- Tokenizer PHP Extension

⁵⁶<https://github.com/laravelista/configure-laravel-5-for-shared-hosting>

⁵⁷<http://studio-renata.hr>

To be honest, I haven't even checked that my shared hosting server has mentioned PHP extensions, but since it all works I guess that it does.

In this tutorial, I will show you how to configure your Laravel application to be able to run on shared hosting. You can view the code for this tutorial on [GitHub](#)⁵⁸.

Moving Files

I have a fresh installation of Laravel on my PC. To see how to install Laravel see the [official documentation](#)⁵⁹ or check out these great courses on my website for [Homestead on Windows](#) or [Laragon](#).



View changes in this commit




















[0546e265b7187312bb360b93edbf91a8e969942](#)⁶⁰.

First, we will organize our application folder structure to better match our needs. The default folder structure looks like this:

⁵⁸<https://github.com/laravelista/configure-laravel-5-for-shared-hosting>

⁵⁹<https://laravel.com/docs/5.2/installation>

⁶⁰<https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/0546e265b7187312bb360b93edbf91a8e969942>

Name	Date modified	Type	Size
 .git	29.7.2016. 13:52	File folder	
 app	29.7.2016. 13:07	File folder	
 bootstrap	29.7.2016. 13:07	File folder	
 config	29.7.2016. 13:07	File folder	
 database	29.7.2016. 13:07	File folder	
 public	29.7.2016. 13:07	File folder	
 resources	29.7.2016. 13:07	File folder	
 storage	29.7.2016. 13:07	File folder	
 tests	29.7.2016. 13:07	File folder	
 vendor	29.7.2016. 13:09	File folder	
 .env	29.7.2016. 13:09	ENV File	1 KB
 .env.example	29.7.2016. 13:07	EXAMPLE File	1 KB
 .gitattributes	29.7.2016. 13:07	Text Document	1 KB
 .gitignore	29.7.2016. 13:07	Text Document	1 KB
 artisan	29.7.2016. 13:07	File	2 KB
 composer.json	29.7.2016. 13:07	JSON File	2 KB
 composer.lock	29.7.2016. 13:07	LOCK File	111 KB
 gulpfile.js	29.7.2016. 13:07	JS File	1 KB
 package.json	29.7.2016. 13:07	JSON File	1 KB
 phpunit.xml	29.7.2016. 13:07	XML File	2 KB
 readme.md	29.7.2016. 13:51	MD File	1 KB
 server.php	29.7.2016. 13:07	PHP File	1 KB

Default folder structure

We will rename folder /public to public_html

You can rename it to whatever you need, but remember to use that name in future code changes in this tutorial.

and create a new directory called `laravel_project`.

You can name this folder whatever you want, but remember to use it in future code references.

Move all files and folders to `laravel_project` directory except `public_html/` and `.git`(if you have created it) folder.

Now, your root folder structure should look like this:

Name	Date modified	Type	Size
.git	29.7.2016. 13:52	File folder	
laravel_project	29.7.2016. 16:47	File folder	
public_html	29.7.2016. 13:07	File folder	
.gitattributes	29.7.2016. 13:07	Text Document	1 KB
.gitignore	29.7.2016. 13:07	Text Document	1 KB

New folder structure 1

and inside laravel_project should look like this:

Name	Date modified	Type	Size
app	29.7.2016. 13:07	File folder	
bootstrap	29.7.2016. 13:07	File folder	
config	29.7.2016. 13:07	File folder	
database	29.7.2016. 13:07	File folder	
resources	29.7.2016. 13:07	File folder	
storage	29.7.2016. 13:07	File folder	
tests	29.7.2016. 13:07	File folder	
vendor	29.7.2016. 13:09	File folder	
.env	29.7.2016. 13:09	ENV File	1 KB
.env.example	29.7.2016. 13:07	EXAMPLE File	1 KB
artisan	29.7.2016. 13:07	File	2 KB
composer.json	29.7.2016. 13:07	JSON File	2 KB
composer.lock	29.7.2016. 13:07	LOCK File	111 KB
gulpfile.js	29.7.2016. 13:07	JS File	1 KB
package.json	29.7.2016. 13:07	JSON File	1 KB
phpunit.xml	29.7.2016. 13:07	XML File	2 KB
readme.md	29.7.2016. 13:51	MD File	1 KB
server.php	29.7.2016. 13:07	PHP File	1 KB

New folder structure 2



View changes in this commit

[6dec090ae1ca11673ee15d7898280becebd46038](https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/6dec090ae1ca11673ee15d7898280becebd46038)⁶¹.

⁶¹<https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/6dec090ae1ca11673ee15d7898280becebd46038>

Update Bootstrap Files Location

Since `index.php` file inside `public_html/` directory is the first file that will be loaded by the HTTP server, we have to adjust the paths to `autoload.php` and `app.php` files located in `/laravel_project/bootstrap/`, to match our new folder structure.

Open file `public_html/index.php` and change:

- **line 22** `require __DIR__.'../bootstrap/autoload.php';` to `require __DIR__.'../laravel_project/bootstrap/autoload.php';`
- **line 36** `$app = require_once __DIR__.'../bootstrap/app.php';` to `$app = require_once __DIR__.'../laravel_project/bootstrap/app.php';`

That is all that you have to change in `public_html` folder.



View changes in this commit

[158c72d065db262ff7139bac9f51178cc5a082a1](https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/158c72d065db262ff7139bac9f51178cc5a082a1)⁶².

Change public_path in Application

So, this party at first seems very scary, but it really isn't. We have created a new class which extends the main Laravel application class and overwrite the `publicPath` method to return the path to our new `public_html` folder. Then we use that class to create a new Laravel application. *This way our public_path works everywhere in our application including console (which you can still use in development).*

Create a new file in `laravel_project/app/` called `Application.php` and paste the following code inside:

Extending the Application class

```
<?php namespace App;
```

```
/**
```

```
 * Extend Laravel main application class to change public_path.
```

```
 *
```

```
 * See here for more info:
```

```
 * http://stackoverflow.com/questions/31758901/laravel-5-change-public-path
```

```
 */
```

⁶²<https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/158c72d065db262ff7139bac9f51178cc5a082a1>

```

class Application extends \Illuminate\Foundation\Application
{
    /**
     * Get the path to the public / web directory.
     *
     * @return string
     */
    public function publicPath()
    {
        return $this->basePath . DIRECTORY_SEPARATOR . '..' . DIRECTORY_SEPARATOR . 'public_html';
    }
}

```

Now in laravel_project/bootstrap/app.php change:

```

$app = new Illuminate\Foundation\Application(
    realpath(__DIR__.'../..')
);

```

to

Using the new Application class

```

$app = new App\Application(
    realpath(__DIR__.'../..')
);

```

This instantiates a new Laravel application using our Application class. Before we can test if our new folder structure works, there is one more step to complete if you plan on using php artisan serve to test the site.



View changes in this commit

[ee254ae1497c739e930c94cd6b85a2c5e25ecf10](https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/ee254ae1497c739e930c94cd6b85a2c5e25ecf10)⁶³.

Make php artisan serve work

This is very simple. Open file laravel_project/server.php and change:

⁶³<https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/ee254ae1497c739e930c94cd6b85a2c5e25ecf10>

```
if ($uri !== '/' && file_exists(__DIR__.'/public'.$uri)) {  
    return false;  
}
```

```
require_once __DIR__.'/public/index.php';
```

to

Updating the path to index.php

```
if ($uri !== '/' && file_exists(__DIR__.'/../public_html'.$uri)) {  
    return false;  
}
```

```
require_once __DIR__.'/../public_html/index.php';
```

Save changes and run

Serving the application

```
cd laravel_project  
php artisan serve
```

You should get output similar to this one:

Development server started

```
$ php artisan serve  
Laravel development server started on http://localhost:8000/
```

If you open your browser on <http://localhost:8000> you should see a standard Laravel Hello page.



Laravel 5

Hello Laravel

You can now compress your application, upload it to your shared hosting and extract. It should work now.



View changes in this commit

[81fd608afe4b551b480640848c28804a5dcc2005](https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/81fd608afe4b551b480640848c28804a5dcc2005)⁶⁴.

This concludes this tutorial.

⁶⁴<https://github.com/laravelista/configure-laravel-5-for-shared-hosting/commit/81fd608afe4b551b480640848c28804a5dcc2005>

The HTTP Layer

Learn how to create and validate forms. Includes tutorials on validating form arrays and different ways of validating data.

Handling Nested Resources

This tutorial will help you understand and handle highly nested resources.

Published at: 14. June, 2016.



View Source Code

Source code for this tutorial is available [here](#)⁶⁵.

This tutorial is a request from a developer colleague. He wanted me to explain how relations work in Laravel and specifically how to handle relationships in highly nested controllers/routes where there are three nested models.

First I will show you what is his actual situation and then I will breakdown it step by step, taking note on what I will need to code the final solution. You can find the repository I created on GitHub.

The solution to this tutorial won't be step by step because showing how to create models, migrations, controllers and routes is not important for this tutorial. The only thing that is important is the controller logic for the show method and displaying the output in the view file. If you are working with nested resources you should already know how to create migrations, models, and controllers.

Scenario

This is his scenario.

He has a URL: `/books/{book_slug}/posts/{post_slug}/comments/{comment_id}` and when that URL is viewed in the browser he wants to display the following:

- Book
 - Name
 - Year
 - Description
 - Number of posts - number of posts that this book contains
- Post
 - Name
 - Description

⁶⁵<https://github.com/laravelista/highly-nested-resources>

- Sum of votes - sum of all votes from comments of this post
- Comment
 - Username - username of the user that voted/commented
 - Body - If the user left a comment body, it should also be displayed
 - Vote - represented in stars; the score this user gave to the post in question
 - Attached files - I won't be covering this because of the length of this tutorial

Calculating Steps

Before doing any actual coding I will first take note on the fields I will need for the models, the models I will need to recreate the scenario and their relationships. I will also take note of what needs to be done to create the desired show method.

Models

From the scenario that I have been given I detect four models in total:

- Book
- Post
- Comment
- User

Now to write down the fields for each model:

- Book
 - name
 - year
 - description
 - slug
- Post
 - name
 - description
 - slug
- Comment
 - body
 - vote
- User
 - name

So the relationships for these models are pretty simple:

- Book:Post 1:N
- Post:Comment 1:N
- User:Comment 1:N

Now in plain English, this means:

- *One* Book can contain *many* Posts, but *one* Post can only belong to *one* Book.
- *One* Post can have *many* Comments, but *one* Comment can only belong to *one* Post.
- *One* User can make *many* Comments, but *one* Comment can belong only to *one* User.

It is all clear here what needs to be done.

For each relationship where the relation is 1 : N, we need to create a foreign key in the model that is marked with N. So to expand our fields:

- Book
 - id - primary key, unsigned int, auto increment
 - name - string
 - year - unsigned int
 - description - string
 - slug - string
- Post
 - id - primary key, unsigned int, auto increment
 - name - string
 - description - string
 - slug - string
 - book_id (foreign_key) - unsigned int
- Comment
 - id - primary key, unsigned int, auto increment
 - body - text
 - vote - unsigned int
 - post_id (foreign_key) - unsigned int
 - user_id (foreign_key) - unsigned int
- User
 - name

Now there is a potential logical issue with the Comment fields regarding the primary key. If we leave the comments table as it currently is, a single user can vote/comment on a post unlimited number of times if a code checking, if the user has voted before, is not set in place. To prevent this behavior we could use a composite primary key that consists of two fields `user_id` and `post_id`. But for this tutorial, I will just let this be since I won't be showing you how to store a comment.

Since Laravel comes with the User model out of the box and the necessary migrations I will use those to save time there.

Controllers

To demonstrate the show method in the above mentioned URL: `/books/{book_slug}/posts/{post_slug}/comments/{comment_id}` I will need just one controller.

There is a convention used for nested controllers where you use resource names in order, to form a controller name. In this case that would be `BookPostCommentController.php`, but you can call it whatever you like or desire. If you don't want to call it `BookPostCommentController.php`, my suggestion is calling it `ShowCommentController.php`.

We can even avoid using a controller completely by using a route closure... I should not have said that :) For code clarity, we will stick with a controller.

Routes

I will only need one route for this:

```
Route::get('books/{book_slug}/posts/{post_slug}/comments/{comment_id}', 'BookPostCommentController@show');
```

Views

We will require only one view file. The tricky question here is where to store that view file. This is how I usually store resource view files:

- books
 - create.blade.php
 - index.blade.php
 - edit.blade.php

I think that placing a folder for each nested resource under `books` should do the trick just nicely, but again you are free to handle your own folder/file management.

- books
 - create.blade.php
 - index.blade.php
 - edit.blade.php
 - posts
 - * comments
 - show.blade.php

To provide some eye candy, I will probably use Bootstrap.

Solution

The solution is pretty simple from this point.

In `app/Http/Controllers/BookPostCommentController.php` we create a show method:

```
public function show($book_slug, $post_slug, $comment_id)
{
    //
}
```

First we find the book by slug:

```
$book = \App\Book::where('slug', $book_slug)->firstOrFail();
```

Then we find the post from that book:

```
$post = $book->posts()->where('slug', $post_slug)->firstOrFail();
```

Finally, we find the comment by comment id from that post:

```
$comment = $post->comments()->findOrFail($comment_id);
```

And we return the view with relevant variables:

```
return view('books.posts.comments.show')
    ->with(compact('book', 'post', 'comment'));
```

The whole show method now looks like this:

```
public function show($book_slug, $post_slug, $comment_id)
{
    $book = \App\Book::where('slug', $book_slug)->firstOrFail();

    $post = $book->posts()->where('slug', $post_slug)->firstOrFail();

    $comment = $post->comments()->findOrFail($comment_id);

    return view('books.posts.comments.show')
        ->with(compact('book', 'post', 'comment'));
}
```

I suggest using eager loading on books query to eager load posts and comments. And then you use collection methods to handle data.

This is the current state of the repository at this moment [0ddc67ed4415d5344b412301b5b300c84b49f67c](https://github.com/laravelista/highly-nested-resources/commit/0ddc67ed4415d5344b412301b5b300c84b49f67c)⁶⁶.

At the link above you can see all things that I had done to the application for this to work.

⁶⁶<https://github.com/laravelista/highly-nested-resources/commit/0ddc67ed4415d5344b412301b5b300c84b49f67c>

Route Caching in Laravel

Quick and painless way to get up to 100x faster route registration in your application.

Published at: 28. September, 2016.

I was a nonbeliever too when I heard about route caching and how it will improve my application loading speed. You are probably reading this tutorial because you want a quick and painless way to gain speed improvements for your application without it being a pain in the ass.

You are lucky. **Route caching is as simple as entering one command and the benefits are awesome.** There is a small chapter on [route caching](#)⁶⁷ in the Laravel 5.3 documentation, so be sure to give it a read. It is very small.

In this tutorial, I will explain what route caching is, how it works and how to use it.

What it is

Route caching is a mechanism which caches your routes file so that that cached file will be loaded on every request instead of scanning the actual routes file and registering all of your application's routes.

“In some cases, your route registration may even be up to 100x faster.” - [source](#)⁶⁸

How it works

Once you run the command, it creates a `routes.php` file in `bootstrap/cache` directory. If you open that file, you should see a comment:

“Here we will decode and unserialize the RouteCollection instance that holds all of the route information for an application. This allows us to instantaneously load the entire route map into the router.” - comment from `bootstrap/cache/routes.php`.

I will not paste the content of the file here because it can be quite huge and it is mostly unreadable to the human eye. This is the gist of that file and what it contains:

⁶⁷<https://laravel.com/docs/5.3/controllers#route-caching>

⁶⁸<https://laravel.com/docs/5.3/controllers#route-caching>

```
<?php

app( 'router' )->setRoutes(
    unserialize(base64_decode('...gibberish...'))
);
```

So basically, route caching loads this file directly into your application on every request instead of scanning the actual routes file and creating a RouteCollection. *This is where the speed benefit comes from.*

How to use it

Now the tricky part. From your project root, in the command line enter:

```
php artisan route:cache
```

And it's done! Simple as that.

If for some reason you want to get rid of route caching, you can run `php artisan route:clear` or manually delete the file `bootstrap/cache/routes.php`.

Remember to generate a fresh route cache every time you add any new routes.

Deployments

If you are using a continuous integration or a continuous deployment service like [Codeship](https://codeship.com/)⁶⁹ or [Envoyer](https://envoyer.io/)⁷⁰ be sure to add `php artisan route:cache` to your deployment commands. This will save you a lot of headaches, trust me.

Troubleshooting

There are two things that could go wrong:

- you forgot to run `php artisan route:cache` when you changed your routes file
- you have closure based routes in your routes file

When in doubt, always run `php artisan route:cache` or `php artisan route:clear`, depending on what you are trying to do.

There is **one limitation of route caching** that I did not mention until now. In order to be able to use route caching, you must convert any closure based routes to controller classes. Basically, no closure routes allowed.

Thank you for reading and I hope that you are using route caching by now.

⁶⁹<https://codeship.com/>

⁷⁰<https://envoyer.io/>

Laravel Forms & HTML

Everything you need to know about working with forms in Laravel. Based on Laravel Collective Forms & HTML package.

Published at: 27. June, 2016.



View Source Code

Source code for this tutorial is available [here](#)⁷¹.

Long time ago, in a galaxy far away, Laravel used to come out-of-the-box with Forms management, but Taylor Otwell (the creator of Laravel) decided that he should focus his time more on core Laravel functionality and that is why he removed Forms from Laravel 5. There were too many issues on forms and it required a lot of time to fix. Instead of fixing issues himself, he decided to invest his time in improving Laravel and that is how the Laravel Collective was born and Forms & HTML package came to be.

In this tutorial, I will show you how to install and use the [Forms & HTML](#)⁷² package from Laravel Collective. We will cover every method mentioned in the documentation including Form Model Binding, Form Model Accessors, Custom Macros, and Components.

Starting Point

I have prepared a Laravel 5.2.37 application on GitHub that you should use as a starting point for this tutorial.

Be sure to read the `readme.md` file that comes with it because you will need to:

- generate app key
- create a database file
- migrate database
- seed the database with fake users

This is the current state of the repository at this moment [915228e0ae082852187b2287914e17127e41b13c](#)⁷³.

⁷¹<https://github.com/laravelista/laravel-forms-and-html>

⁷²<https://laravelcollective.com/docs/5.2/html>

⁷³<https://github.com/laravelista/laravel-forms-and-html/commit/915228e0ae082852187b2287914e17127e41b13c>

Installation

To install Forms & HTML package from the command line type:

```
composer require laravelcollective/html
```

Output:

```
Using version ^5.2 for laravelcollective/html
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing laravelcollective/html (v5.2.4)
  Downloading: 100%
```

Now we have to add the service provider to our application in config/app.php:

```
'providers' => [
    ...
    Collective\Html\HtmlServiceProvider::class
];
```

And finally let's add a facade aliases for it:

```
'aliases' => [
    ...
    'Form' => Collective\Html\FormFacade::class,
    'Html' => Collective\Html\HtmlFacade::class
];
```

You will most likely not need the `Html` facade. Everything you need for working with forms is located in `Form` facade.

Form basics

There are some basic things about working with forms. This is the stuff that you need to be aware of before creating a form in your application. You need to know how to open a form for specific HTTP method, point the form where you want, add support for files, bind a model to a form and handle custom model field output. *This last one is optional, but has a lot of benefits.*

I have created a view in `resources/views` folder called `forms.blade.php` and there I have included bootstrap CSS and JS files from the CDN to provide some eye candy to our form. In the `app/Http/routes.php` file I have replaced the default view template from `welcome` to `forms` in order to show the newly created `forms.blade.php` view. All code examples mentioned bellow will be added in that file.

```
Route::get('/', function () {
    return view('forms');
});
```

This is the current state of the repository at this moment [de2644df4eede7bfea0c0b1fa861d97236db8ba4](https://github.com/laravelista/laravel-forms-and-html/commit/de2644df4eede7bfea0c0b1fa861d97236db8ba4)⁷⁴.

Opening A Form

The most simple code to open a form looks like this:

```
{{ Form::open(['url' => '/']) }}

{{ Form::close() }}
```

By default, if not specified differently, a POST method is assumed on the form. You can use POST, GET, PUT, DELETE methods, but since HTML forms only support POST and GET methods, all other methods will be spoofed automatically by adding a `_method` hidden field to your form.

If you check the page source code in your browser now, you will see:

```
<form method="POST" action="http://localhost:8000" accept-charset="UTF-8">
  <input name="_token" type="hidden" value="OVxiaKOIQev7E799PUDhfwilA5NoWeL8Nz\
6cPutZ">
</form>
```

The hidden `_token` field you see in the form is used for CSRF protection which will be discussed in the chapter bellow.

Changing Form Method

To change the form method, just add a key/value to the form parameters like so:

⁷⁴<https://github.com/laravelista/laravel-forms-and-html/commit/de2644df4eede7bfea0c0b1fa861d97236db8ba4>

```
{{ Form::open(['url' => '/', 'method' => 'GET']) }}

{{ Form::close() }}
```

Using Route Name

Instead of using 'url' => '/' to set the action of your form, you can also use route names or controller actions. *I always prefer using route names.*

I have updated app/Http/routes.php file so that our only route is now named home:

```
Route::get('/', ['as' => 'home', function () {
    return view('forms');
}]);
```

To use a route name instead of url when opening a form, use it like this:

```
{{ Form::open(['route' => 'home']) }}

{{ Form::close() }}
```

Using Controller Actions

To demonstrate using controller actions in forms I will need to create a controller and add a route for it. Let's create a resource controller called UsersController using artisan from the command line:

```
php artisan make:controller UsersController --resource
```

And now add a route in app/Http/routes.php:

```
Route::resource('users', 'UsersController');
```

To set the action of the form using a controller action, use this code:

```
{{ Form::open(['action' => 'UserController@store']) }}

{{ Form::close() }}
```

Passing Route Parameters

When using route names or controller actions to set the action of the form, sometimes you need to pass route parameters, like the current resource id. This happens in cases when you edit, update or destroy a specific resource.

To pass route parameters use an array:

```

{{ Form::open(['route' => ['users.show', 1]]) }}
{{ Form::close() }}

{{ Form::open(['action' => ['UsersController@show', 1]]) }}
{{ Form::close() }}

```

Accept file uploads

If your form is going to accept file uploads, add a `files` key to the form parameters:

```

{{ Form::open(['route' => 'home', 'files' => true]) }}

{{ Form::close() }}

```

Setting the `files` key value to `true` adds `enctype="multipart/form-data"` to the form tag. *Without this your form will not upload files.*

This is the current state of the repository at this moment [b23012be7dde07c21b2d71cc4d96e75393fd3f66](https://github.com/laravelista/laravel-forms-and-html/commit/b23012be7dde07c21b2d71cc4d96e75393fd3f66)⁷⁵.

CSRF Protection

By default, if you use `Form::open` (as we are in this tutorial) to create your form the CSRF token will be added as a hidden field automatically. If for some reason you are still opening a form using HTML tag form you will have to add the field manually. You can do that in two ways:

1. Place `{{ Form::token() }}` inside the form or
2. Use Laravel helper method `{{ csrf_field() }}`

Hint! By default, all routes are CSRF protected, if you want to add exceptions go to `app/Http/Middleware/VerifyCsrfToken.php` file and add the URIs you want to exclude from CSRF protection there.

This is the current state of the repository at this moment [9800a7b53c2beb8dcdf0a2feeff3fa7bbd089228](https://github.com/laravelista/laravel-forms-and-html/commit/9800a7b53c2beb8dcdf0a2feeff3fa7bbd089228)⁷⁶.

⁷⁵<https://github.com/laravelista/laravel-forms-and-html/commit/b23012be7dde07c21b2d71cc4d96e75393fd3f66>

⁷⁶<https://github.com/laravelista/laravel-forms-and-html/commit/9800a7b53c2beb8dcdf0a2feeff3fa7bbd089228>

Form Model Binding

Now, this is something pretty cool and if you are not using it, you better start right away. It saves you so much time when dealing with populating a form from an existing model, validating form data and setting default values.

When using Form Model Binding, the form is automatically populated with that model values if the form field name matches the model attribute name. Also if there is session flash data, that data takes precedence over the model's value. This is the priority:

1. Session Flash Data (Old Input)
2. Explicitly Passed Values
3. Model Attribute Data

This is very helpful when dealing with validation errors on the server.

I will guide you through the process of populating a form with an existing user from our database. We want to create a page where the user can edit his own profile information (name, email, password). Since we already have the routes and controller set up for the users resource we will change the edit method in `app/Http/Controllers/UsersController.php` to find the user by id and to return a view containing the form, which we will later create.

```
$user = \App\User::findOrFail($id);

return view('users.edit')->with(compact('user'));
```

This is the current state of the repository at this moment [6eb2993effd0b1455e66e35533b5fedc06036232](https://github.com/laravelista/laravel-forms-and-html/commit/6eb2993effd0b1455e66e35533b5fedc06036232)⁷⁷.

I have implemented a layout view file to ease the process of creating a new view file with bootstrap dependencies.

Now let's create a view in `resources/views/users` called `edit.blade.php` and inside it, paste the following:

```
@extends('layout')

@section('content')
    {{-- future markup goes here --}}
@stop
```

What do we want to do now?

When we visit URI `users/1/edit`, we want to be shown a form populated with that users name, email and a blank field for password and password confirmation. Let's do that now:

⁷⁷<https://github.com/laravelista/laravel-forms-and-html/commit/6eb2993effd0b1455e66e35533b5fedc06036232>

```

{{ Form::model($user, [
    'method' => 'PUT',
    'route' => ['users.update', $user->id]
]) }}

<div class="form-group">
    {{ Form::label('name', 'Name') }}
    {{ Form::text('name', null, ['class' => 'form-control']) }}
</div>

<div class="form-group">
    {{ Form::label('email', 'Email') }}
    {{ Form::email('email', null, ['class' => 'form-control']) }}
</div>

<div class="form-group">
    {{ Form::label('password', 'Password') }}
    {{ Form::password('password', ['class' => 'form-control']) }}
</div>

<div class="form-group">
    {{ Form::label('password_confirmation', 'Password Confirmation') }}
    {{ Form::password('password_confirmation', ['class' => 'form-control']) }}
</div>

{{ Form::submit('Update', ['class' => 'btn btn-default']) }}

{{ Form::close() }}

```

Note! When using `Form::model`, be sure to close your form with `Form::close`.

For now, ignore Form package code. I just want you to see that the fields we defined will be automatically populated with that model values. When you visit URI `/users/1/edit` you will see a form that looks something like this:

Name

Email

Password

Password Confirmation

Populated edit form

As you can see, the name and email fields are automatically populated and password and password_confirmation fields are left empty. Try entering some value in password field, change the value of name field and submit the form. You will get validation alert and the value you entered in name field is preserved.

In many cases, the same form can be reused for edit and create actions by using blade template partials. This saves you a lot of time.

This is the current state of the repository at this moment [d642f5d035436b8e30edd89532da2c463fd808](https://github.com/laravelista/laravel-forms-and-html/commit/d642f5d035436b8e30edd89532da2c463fd808)⁷⁸.

Form Model Accessors

Now this one is a bit tricky to explain, especially to someone who doesn't know what [eloquent accessors](#)⁷⁹ are, but here goes something. Laravel's Eloquent accessors allow you to manipulate a model attribute before returning it. This is useful in cases where you want to globally return preformatted values, but sometimes you just want to return different data in form elements.

Trust me, this will happen to you at some point and it is wise to remember that this chapter exists when working with forms.

To demonstrate this, I will use the previous form we used for editing a user model. Let's say that for some reason you want to populate the users name field with uppercase value of user name attribute. (I really don't know why someone would want to do this, but what the hell.)

There is nothing that we have to do to our form to change the way our form gets the name value from the user model. We only need to add one method to our App\User model and add a trait. Open app/User.php and add the following method:

⁷⁸<https://github.com/laravelista/laravel-forms-and-html/commit/d642f5d035436b8e30edd89532da2c463fd808>

⁷⁹<http://laravel.com/docs/5.2/eloquent-mutators#accessors-and-mutators>


```
public function formNameAttribute($value)
{
    return strtoupper($value);
}
```

and inside the User class declaration add the following trait:

```
use \Collective\Html\Eloquent\FormAccessible;
```

Now when you open the same form as in the previous chapter located in `users/1/edit` you will see that the user's name is in uppercase. You can apply the same logic to all fields that you need to change.

This is the current state of the repository at this moment [c57e8a248aa005be12af91c67bcfe13c4ebf94b7](https://github.com/laravelista/laravel-forms-and-html/commit/c57e8a248aa005be12af91c67bcfe13c4ebf94b7)⁸⁰.

Input basics

To demonstrate different types of input fields we will create a new route that will return a view called `inputs.blade.php`. To create this, first add a route in our `app/Http/routes.php` file:

```
Route::get('/inputs', function () {
    return view('inputs');
});
```

Then create a new file in `resources/views/` called `inputs.blade.php` and paste the following inside:

```
@extends('layout')

@section('content')

    {{ Form::open(['url' => '/']) }}

    {{-- Future code goes here. --}}

    {{ Form::close() }}

@stop
```

To view code examples mentioned bellow, navigate your Internet browser to URI `/inputs`.

This is the current state of the repository at this moment [da1af87259662150964d0f23356e4ea6c27d215e](https://github.com/laravelista/laravel-forms-and-html/commit/da1af87259662150964d0f23356e4ea6c27d215e)⁸¹.

Let's move on.

⁸⁰<https://github.com/laravelista/laravel-forms-and-html/commit/c57e8a248aa005be12af91c67bcfe13c4ebf94b7>

⁸¹<https://github.com/laravelista/laravel-forms-and-html/commit/da1af87259662150964d0f23356e4ea6c27d215e>

Labels

To generate a Label element use this syntax:

```
{{ Form::label('email', 'E-mail Address') }}
```

You can specify extra HTML attributes by using an array:

```
{{ Form::label('email', 'E-mail Address', ['class' => 'some-sick-label']) }}
```

Text, Text Area, Password & Hidden Fields

To generate a Text Input element use this syntax:

```
{{ Form::text('username') }}
```

To specify a default value use this:

```
{{ Form::text('username', 'default-value') }}
```

To generate a hidden input element use this:

```
{{ Form::hidden('something_fancy') }}
```

To generate a textarea input element use this:

```
{{ Form::textarea('message') }}
```

You can also pass the default value to all text input element as in `Form::text` example. Furthermore, you can also specify a third parameter to add extra HTML attributes like so:

```
{{ Form::textarea('message', 'default message', ['cols' => 15]) }}
```

To generate a password input element use this:

```
{{ Form::password('password', ['class' => 'form-control']) }}
```

You can omit the array since it specifies a class to be applied to the element.

To generate an email input element use:

```

{{ Form::email($name, $value = null, $attributes = []) }}
// or
{{ Form::email('email', null, []) }}
// or
{{ Form::email('email') }}

```

To generate a file input element use this:

```

{{ Form::file('profile_image', ['class' => 'something-pretty']) }}

```

The form must have been opened with the files option set to true.

As in previous examples, you can omit the array if you don't need to pass extra HTML attributes.

This is the current state of the repository at this moment [849b07169ab193230892e29d8e33744a868eed2c](https://github.com/laravelista/laravel-forms-and-html/commit/849b07169ab193230892e29d8e33744a868eed2c)⁸².

Checkboxes and Radio Buttons

To generate a checkbox or a radio button use this:

```

{{ Form::checkbox('name', 'value') }}

{{ Form::radio('name', 'value') }}

```

To mark checkbox or radio button as checked, set the third parameter to true, like so:

```

{{ Form::checkbox('name', 'value', true) }}

```

This is the current state of the repository at this moment [fc1862053064ad838ed3bf2d8f5cb23304927ab3](https://github.com/laravelista/laravel-forms-and-html/commit/fc1862053064ad838ed3bf2d8f5cb23304927ab3)⁸³.

Number

To generate a number input use this:

```

{{ Form::number('name', 'value') }}

```

Date

To generate a date input use this:

⁸²<https://github.com/laravelista/laravel-forms-and-html/commit/849b07169ab193230892e29d8e33744a868eed2c>

⁸³<https://github.com/laravelista/laravel-forms-and-html/commit/fc1862053064ad838ed3bf2d8f5cb23304927ab3>

```
{{ Form::date('name', \Carbon\Carbon::now()) }}
```

This is the current state of the repository at this moment [e9508df184d00a11299f44989f50d4d6e384406e](https://github.com/laravelista/laravel-forms-and-html/commit/e9508df184d00a11299f44989f50d4d6e384406e)⁸⁴.

Drop-Down Lists

To generate a drop-down list use this:

```
{{ Form::select('size', ['L' => 'Large', 'S' => 'Small']) }}
```

To generate a drop-down list with a selected default value use this:

```
{{ Form::select('size', ['L' => 'Large', 'S' => 'Small'], 'S') }}
```

To generate a drop-down list with placeholder instead of default value:

```
{{ Form::select('size', ['L' => 'Large', 'S' => 'Small'], null, ['placeholder' => 'Pick a size']) }}
```

To generate a grouped list:

```
{{ Form::select('animal', [
    'Cats' => ['leopard' => 'Leopard'],
    'Dogs' => ['spaniel' => 'Spaniel'],
]) }}
```

To generate a drop-down list with a range:

```
{{ Form::selectRange('number', 10, 20) }}
```

To generate a drop-down list with month names:

```
{{ Form::selectMonth('month') }}
```

This is the current state of the repository at this moment [06caff3a49d460d7fc9a50b8946f73023d1a8b43](https://github.com/laravelista/laravel-forms-and-html/commit/06caff3a49d460d7fc9a50b8946f73023d1a8b43)⁸⁵.

Buttons

To generate a submit button use this:

⁸⁴<https://github.com/laravelista/laravel-forms-and-html/commit/e9508df184d00a11299f44989f50d4d6e384406e>

⁸⁵<https://github.com/laravelista/laravel-forms-and-html/commit/06caff3a49d460d7fc9a50b8946f73023d1a8b43>

```
{{ Form::submit('Send') }}
```

or to generate a regular button, you can use this:

```
{{ Form::button('Click me!', ['class' => 'btn btn-primary']) }}
```

Custom Macros

Now, this is something really cool, but I must admit that I rarely use it. Macros enable you to define your own “shortcuts” to use in your forms.

First, you have to register a macro. The place that makes most sense for this is in a new service provider, but for the purpose of this tutorial we will place it in `app/Providers/AppServiceProvider.php` file.

Open the above-mentioned file and in `boot` method paste the following:

```
public function boot()
{
    \Form::macro('superSecretHiddenField', function()
    {
        return '<input type="hidden" value="secret">';
    });
}
```

This enables us to call our macro from any view. Now, go back to `resources/views/inputs.blade.php` and call our macro from there:

```
{!! Form::superSecretHiddenField() !!}
```

This is the current state of the repository at this moment [19a3c9fea52538c0f5205c79742a42a9ac10e036](https://github.com/laravelista/laravel-forms-and-html/commit/19a3c9fea52538c0f5205c79742a42a9ac10e036)⁸⁶.

Custom Components

Same as macros, but even better and again I barely use this feature, but I find it very cool. In short, you define your own custom form element as a component using a blade view file instead of a closure as we did with form macros. This is very useful when you are using a front-end framework like Twitter Bootstrap.

Let me show you how you would define a twitter bootstrap formatted form component for text input.

Let's start by registering our component in `app/Providers/AppServiceProvider.php`. In `boot` method add the following:

⁸⁶<https://github.com/laravelista/laravel-forms-and-html/commit/19a3c9fea52538c0f5205c79742a42a9ac10e036>

```
\Form::component('textField', 'components.form.text', ['name', 'value' => null, \
'attributes' => []]);
```

components.form.text defines where the view to render this component is located. We need to create a folder in resources/views/ called components and inside that folder create another one called form. Inside that folder create an empty blade file and call it text.blade.php. Paste the following inside the file:

```
<div class="form-group">
    {{ Form::label($name, ucfirst($name)) }}
    {{ Form::text($name, $value, array_merge(['class' => 'form-control'], $attributes)) }}
</div>
```

To call our custom component, go to resources/views/inputs.blade.php and at the bottom add:

```
{{ Form::textField('name') }}
```

This will render in HTML:

```
<div class="form-group">
    <label for="name">Name</label>
    <input class="form-control" name="name" type="text" id="name">
</div>
```

When you are working on a big project with a group of people this stuff is very handy to have because everyone in your team then produces the same form elements.

I personally have set up snippets using Sublime Text and they work fine for me.

This is the current state of the repository at this moment [6f5a222ef2ef4c4a2b01cba3a8753ce052ea833e](https://github.com/laravelista/laravel-forms-and-html/commit/6f5a222ef2ef4c4a2b01cba3a8753ce052ea833e)⁸⁷.

This concludes this tutorial.

⁸⁷ <https://github.com/laravelista/laravel-forms-and-html/commit/6f5a222ef2ef4c4a2b01cba3a8753ce052ea833e>

Different ways of validating requests

Laravel provides three different ways of validating data from a request. In this tutorial, we will go through all three ways, to find out which approach is most suitable for a specific situation.

Published at: 13. January, 2017.

We will start this new year with a free tutorial, aimed at beginners and newcomers to the framework. I hope that this tutorial will clarify things for you, and help you choose a validation approach for your own Laravel application.

I suggest you read all the documentation on [Validation](#)⁸⁸ in Laravel. The *Validation Quickstart* chapter contains everything you need to know to get started. Also, read the *Available Validation Rules* chapter to get to know all the different ways in which you can validate data.

Controller validation

This is one of the reasons why I love Laravel. In your controller methods you can inject `\Illuminate\Http\Request $request` and then from the method call `$this->validate` method and pass it the current `$request` with the validation rules array.

```
$this->validate($request, [
    'field' => 'rule|rule|rule'
]);
```

It's simple as that. I think that this is the most used way to validate data in a controller.

The benefit of using this approach is the speed of implementation. The drawback only hits you much later when you have to validate the same resource again in a different place.

Ease of use 3/3 <div class="progress"> <div class="progress-bar" style="width: 100%;"></div> </div>

Pretty code 2/3 <div class="progress"> <div class="progress-bar" style="width: 60%;"></div> </div>

This is the easiest way to handle data validation. The validation code that you write will be visible in the controller method, but it looks pretty.

⁸⁸<https://laravel.com/docs/5.3/validation>

Form Request Validation

There are two reasons for using form request classes. First, because your controller methods will look slim and second because you can reuse them in different parts of your application.

The drawback of this approach is that it takes you a few extra steps to get it working.

You have to create a form request using `php artisan make:request CreateUser` command and then write your validation rules inside that class, along with handling the `authorize` method. Then, you have to replace `Request $request` from your controller method that handles data validation, with your new form request class `App\Http\Requests\CreateUser $request`. That's it. There is nothing else that you need to write in your method. The validation is handled automatically.

Ease of use 2/3 `<div class="progress"> <div class="progress-bar" style="width: 60%;"></div> </div>`

Pretty code 3/3 `<div class="progress"> <div class="progress-bar" style="width: 100%;"></div> </div>`

I like this approach very much, but the reason why I don't always use it is because of those few extra steps. Usually what I do is use the `$this->validate` method and then later when I need to reuse validation, I create a form request class and move the validation rules there.

Manually Creating Validators

This is the way in which we have handled data validation in Laravel 4. I really do my best to avoid using this approach. But, sometimes this approach makes the most sense.

```
$validator = Validator::make($request->all(), [
    'title' => 'required|unique:posts|max:255',
    'body' => 'required',
]);
```

I use this approach in edge cases. When I need to do something different than returning a 422 JSON response or redirecting back with errors. You can use `if($validator->fails())` method to check if the validation has failed and then do an appropriate action.

Ease of use 1/3 `<div class="progress"> <div class="progress-bar" style="width: 30%;"></div> </div>`

Pretty code 1/3 `<div class="progress"> <div class="progress-bar" style="width: 30%;"></div> </div>`

Another use case for this is when you need to validate some data, but you are not in the controller. You can pass it an array of data and it will validate it. It can be very useful sometimes.

I hope that you have enjoyed this little tutorial.

Validating Form Arrays

Handling form array validation is no longer a pain in the ass, as it was with Laravel 4.

Published at: 13. September, 2016.



View Source Code

Source code for this tutorial is available [here](#)⁸⁹.

It happened to me many times before. I would write a sick new [React.js](#)⁹⁰ component or glued together a quick [jQuery](#)⁹¹ script to enable users to add multiple instances of the same type on one form and then the dilemma would occur: “Should I just add array in validation rules for that field or would I create a custom validation rule for that specific type of field?”

The latest form that I am currently working on has the ability to add multiple device groups in which you can select the device type and enter the quantity for that device type. Meaning that I would have to have a `exists` validation rule for every device type field and an integer check for quantity with some other validation rules.

In this tutorial I will skip the JS part of generating the form and will show you how the form looks like, how the HTML should be formatted, how to write validation rules for form array and some extra tips on how to avoid multiple `exists` calls to the database that could slow down your application.

Build your Form

This is how the form that I will be using to demonstrate form array validation looks like:

⁸⁹<https://github.com/laravelista/validating-form-arrays>

⁹⁰<https://facebook.github.io/react/>

⁹¹<https://jquery.com/>

Winners & prizes

Choose winners and enter the prizes that they have won.

Winner

Destiney Olson Sr.

Prize

Enter prize

Winner

Dr. Jensen Macejkovic

Prize

Enter prize

[Add another winner](#)

Form

You can visit the repository for this tutorial on [GitHub](#).

I created a form in which you add lottery winners. For each winner, you have to select the user that won and enter the prize that he has won. *This is all for demonstration purposes only*. When you press the “Add another winner” button a new winner group is added to the page with Javascript (This has not been implemented in the repository. Feel free to send a PR).

This is what the generated HTML looks like:

```
<form method="POST" action="http://localhost:8000">

<!-- Generated with Javascript -->
<div class="winner">
  <div class="form-group">
    <label for="winner[1][user_id]">Winner</label>
    <select class="form-control" name="winner[1][user_id]">
      <option value="1">Destiney Olson Sr.</option>
      <option value="5">Dr. Jensen Macejkovic</option>
      <option value="4">Eleazar Gulgowski PhD</option>
    </select>
  </div>
  <div class="form-group">
    <label for="winner[1][prize]">Prize</label>
    <input type="text" class="form-control" name="winner[1][prize]" placeholder="Enter prize">
  </div>
</div>
```

```

    <hr />
</div>

<!-- Generated with Javascript -->
<div class="winner">
    <div class="form-group">
        <label for="winner[2][user_id]">Winner</label>
        <select class="form-control" name="winner[2][user_id]">
            <option value="1">Destiney Olson Sr.</option>
            <option value="5">Dr. Jensen Macejkovic</option>
            <option value="4">Eleazar Gulgowski PhD</option>
        </select>
    </div>
    <div class="form-group">
        <label for="winner[2][prize]">Prize</label>
        <input type="text" class="form-control" name="winner[2][prize]" placeholder="Enter prize">
    </div>
    <hr />
</div>

<a href="#" class="btn btn-link">Add another winner</a>

<div class="form-group">
    <br />
    <button type="submit" class="btn btn-primary btn-lg">Submit</button>
</div>

</form>

```

Things to keep in mind:

- the name attribute must/should follow this pattern `group_name[unique_number][field_name]`
- `unique_number` must be unique in that `group_name` to avoid overwriting fields

I must admit that I have been doing some crazy stuff when working with form arrays. One thing that I would do is to have fields with `field_name[]` and `related_field_name[]` and then I would manually loop both arrays and match both fields by the index in the loop. Don't do this :)

Once you have your form names in the correct format, validation is very simple since Laravel 5.2.

Write Validation Rules

First I will show you the entire code needed to validate the above form and then I will explain it step by step.

```
public function store(Request $request)
{
    $this->validate($request, [
        'winner.*.user_id' => 'bail|required|integer|min:1|exists:users,id',
        'winner.*.prize' => 'required|string|max:255'
    ]);

    return "Validation was successful.";
}
```

As you can see, to validate form arrays we use the `*` symbol, followed by the `field_name` like so `group_name.*.field_name`. The first validation rule is for the `user_id` field inside the form array. We want to be sure that the `user_id` that has been sent to the server inside the form array actually exists in our database, but to decrease the potential amount of queries on the database that that validation rule can generate I added a few checks before the `exists` rule.

First, we tell the Validator to stop on first validation failure with `bail` rule. Then we tell it that this field is required and that it should be an integer. Since we know that all IDs start with 1, in this case, we can safely add a `min` rule to be 1. Finally, if all previous rules have passed we check the database if the given id exists.

This is a small potential speed improvement trick. It does not hurt to implement it, but you can survive with it in most cases.

The second field we validate is the `prize` field. This is pretty straightforward, we need it to be required, string and the max number of characters should be 255 because of the `VARCHAR` max limit.

If all rules passed you should see a message `Validation was successful.` on the page. Perfect!

This is the current state of the repository at this moment [4cceca579527ef4f15b5ed84de37196f257ba58b](https://github.com/laravelista/validating-form-arrays/commit/4cceca579527ef4f15b5ed84de37196f257ba58b)⁹².

This concludes this tutorial.

P.S. Send me a PR that handles adding a new “winner” group. The best PR will be merged and author will be credited. Best of luck and remember simpler is better.

⁹²<https://github.com/laravelista/validating-form-arrays/commit/4cceca579527ef4f15b5ed84de37196f257ba58b>

Creating Custom Validation Rules

If validation rules that come with Laravel are not enough for you, then you can create your own validation rules. How awesome is that!

Published at: 27. March, 2017.



View Source Code

Source code for this tutorial is available [here](#)⁹³.

This is a tutorial that I wanted to write for a very long time.

The reason why I didn't is because I couldn't imagine a single rule that could be beneficial and does not already come with Laravel.

Are you ready?

In this tutorial, we will create a custom validation rule for:

```
if($number % 2 == 0) return true;
```

\$number being the value passed in the request, and 2 being the parameter that we want to divide it by.

The % operator computes the remainder after dividing its first operand by its second.

The above code will return true if the remainder equals zero.

This would be the desired end syntax for our validation rule:

```
$this->validate($request, [  
    'number' => 'required|numeric|mod:2'  
]);
```

P.S. If you want to learn more about standard Validation, there are a couple of tutorials on this site that cover that topic:

- [Different ways of validating requests](#)
- [Elementary Laravel: Validation](#)
- [Validating Form Arrays](#)

⁹³<https://github.com/laravelista/custom-validation-rules>

Hack all day, hack all night

Let's begin.

There are two ways of registering custom validation rules:

1. Using the extend method on the Validator facade
2. Using the extendImplicit method on the Validator facade

The extendImplicit method enables you to run your custom validation rule on a empty attribute (The value is null). By default, rules are not executed on attributes that have a value of null. *We won't be needing this for this tutorial, but it is good to know.*

Both extend and extendImplicit methods support two ways of registering rules:

Using a Closure

```
Validator::extend('foo', function ($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

or

Passing a class and method

```
Validator::extend('foo', 'FooValidator@validate');
```

For this tutorial we will use a Closure.

Using a Closure

```
Validator::extend('foo', function ($attribute, $value, $parameters, $validator) {
    return $value == 'foo';
});
```

The custom validator Closure receives four arguments: the name of the \$attribute being validated, the \$value of the attribute, an array of \$parameters passed to the rule, and the Validator instance.

So to modify the code above to our needs we will want to do:

```
\Validator::extend('mod', function ($attribute, $value, $parameters, $validator)\
{
    return $value % $parameters[0] == 0;
});
```

Great! Now paste the above code to your app/Providers/AppServiceProvider@boot method.

Validate this!

You can use our newly created validation rule from your controller methods, like so:

```
$this->validate($request, [
    'number' => 'required|numeric|mod:2'
]);
```

or from your routes/web.php routes file, like so:

```
Route::get('/', function (\Illuminate\Http\Request $request) {
    $validator = \Validator::make($request->all(), [
        'number' => 'required|numeric|mod:2'
    ]);

    if($validator->fails())
    {
        return $validator->errors();
    }

    return "Validation passed!!";
});
```

Defining the error message

To provide a pretty error message for our custom validation rule, you should add:

```
"mod" => "Your input was invalid!",
```

to the first level of the array in resources/lang/en/validation.php.

You can see the whole code for this tutorial on [GitHub](https://github.com/laravelista/custom-validation-rules)⁹⁴. Or just the code for this tutorial [here](https://github.com/laravelista/custom-validation-rules/commit/42637b403342bd01cd474bb5a36edff2f5875695)⁹⁵.

Thank you for reading!

⁹⁴<https://github.com/laravelista/custom-validation-rules>

⁹⁵<https://github.com/laravelista/custom-validation-rules/commit/42637b403342bd01cd474bb5a36edff2f5875695>

Frontend: Compiling Assets

Learn how to use Laravel Mix to compile frontend assets. Also, learn how to use Laravel Mix or Laravel Elixir without Laravel.

Sublimely Magnificent Laravel Mix

A remake of Laravel Elixir is now called Laravel Mix. It comes with Laravel 5.4 out-of-the-box and in this free tutorial, we will go through all the important parts so that you can quickly get started with it.

Published at: 16. February, 2017.

In plain terms, the main difference between Laravel Elixir and Laravel Mix is that Laravel Mix is based on [Webpack](#)⁹⁶, while Laravel Elixir is based on [Gulp](#)⁹⁷. But, under the hood Laravel Mix is a much more elegant and cleaner solution.

In this tutorial, we will go through all the basics like:

- installation
- running
- stylesheets (source maps)
- JavaScript (code splitting)
- copying
- versioning
- notifications

The main reason why I think that Laravel Mix is better than Laravel Elixir is because it does not assume the folder structure. You reference the resource files from the root directory, and your folder structure (input & output directories) can be whatever you want.

This means that you can use Laravel Mix outside a Laravel application without any additional configuration like we had to do with Laravel Elixir in my recent tutorial [Laravel Elixir Without Laravel](#).

Another very cool feature of Laravel Mix is **Code splitting**. Continue reading to find out more.

Installation

Requirements for installing Laravel Mix are Node.js and NPM. You can verify that you have them installed by running these commands in your command line:

⁹⁶<https://webpack.js.org/>

⁹⁷<http://gulpjs.com/>

```
node -v
npm -v
```

If you do not have them installed, you can download the installers from the [Node.js⁹⁸](#) download page.

Because Laravel Homestead comes with Node.js and NPM preinstalled, a lot of you may be tempted to use it for asset manipulation. I suggest avoiding doing that, and instead, I suggest that you install Node.js and NPM on your host machine; in my case Windows PC.

Within a fresh installation of Laravel 5.4 comes a `package.json` file and inside it, you will find the following dependencies:

```
{
  ...
  "devDependencies": {
    "axios": "^0.15.3",
    "bootstrap-sass": "^3.3.7",
    "jquery": "^3.1.1",
    "laravel-mix": "^0.7.2",
    "lodash": "^4.17.4",
    "vue": "^2.1.10"
  }
}
```

If you read the documentation on Laravel Mix, it says that to install Laravel Mix, you just have to enter `npm install` in your command line which is false. Running `npm install` at this point will install the above mentioned “other” stuff, which if you are like me You don’t actually need.

Laravel Mix does not need any of the other dependencies mentioned in that file to work. Those are some Laravel specifics which you probably don’t need. I personally just delete them from `package.json` file, and leave only `laravel-mix`, so that my file looks like this:

⁹⁸<https://nodejs.org/en/download/>

```
{
  ...
  "devDependencies": {
    "laravel-mix": "^0.7.2"
  }
}
```

Now run `npm install` to install Laravel Mix.

One thing to mention. In the Laravel Mix documentation it says that if you are running a VM on a Windows host system, you may need to run `npm install --no-bin-links`. **Don't do it!**

If you have Node.js and NPM installed on Windows, and are using Git Bash (which comes with Git; with symbolic links enabled) then everything will work correctly. Using `--no-bin-links` just makes a huge mess on the disk and takes much more disk space.

Running

If you take a look in `package.json` file, you should see four scripts mentioned in that file:

- `dev`
- `watch`
- `hot`
- `production`

`npm run dev` - runs all Mix tasks. One time thing.

`npm run watch` - runs all Mix tasks and continues to watch your assets for changes. If a change occurs it automatically recompiles your assets. Use this while actively working on your assets.

`npm run production` - runs all Mix tasks and minifies output. When you are done working on your assets, run this command before pushing your application to production.

`npm run hot` - this is complicated to explain and I am not sure that I know exactly how it works, but here it goes. It runs all Mix tasks, but it stays active and watches for changes in your assets. If a change occurs it “updates” only the module which has changed, notifies your application of the change and updates the code in the browser without a refresh. Something like that... very cool!

Stylesheets (source maps)

`webpack.mix.js` file is the main file for Laravel Mix in which all of your tasks are to be defined.

Laravel Mix supports Less, Sass, plain CSS and Source Maps.

Less & Sass

The very basic example is this:

```
mix.less('resources/assets/less/app.less', 'public/css');
```

It compiles `resources/assets/less/app.less` into `public/css/app.css`.

You can modify the output file name and/or location by specifying it in the second argument like so:

```
mix.less('resources/assets/less/app.less', 'public/stylesheets/styles.css');
```

To compile multiple Less files you can chain the tasks together like so:

```
mix.less('resources/assets/less/app.less', 'public/css')  
    .less('resources/assets/less/admin.less', 'public/css');
```

The same things apply to Sass tasks, namely `.scss` and `.sass` files. Just replace `mix.less` with `mix.sass` and you are ready to go.

Plain CSS

To combine multiple CSS files into a one file, you may use the `mix.combine` task like so:

```
mix.combine([  
    'public/css/vendor/normalize.css',  
    'public/css/vendor/videojs.css'  
], 'public/css/all.css');
```

Source Maps

By default, source maps are disabled because they make the whole process of compiling assets longer and more resource demanding. I personally like using Source Maps because I find it very helpful to be able to inspect an element in the browser, and see exactly on which line and in which file is that style or class applied.

To enable Source Maps, you have to add `mix.sourceMaps()` to your `webpack.mix.js` file:

```
mix.js('resources/assets/js/app.js', 'public/js')
    .sourceMaps();
```

JavaScript (code splitting)

Running this single line of code:

```
mix.js('resources/assets/js/app.js', 'public/js');
```

takes care of:

- compiling ECMAScript 2015
- module bundling
- minification
- concatenating plain JavaScript files

Very cool stuff, but the feature that I find the most interesting is **Code Splitting**.

Code Splitting

So imagine that you are working on your application JS code and you import jQuery, React.js, Vue.js and many other “vendor” libraries. And besides that, you have your own JS code written in modules.

If you would to just use `mix.js` to compile your JS files, you would get a huge file containing all the code necessary for execution. Lets say that that file is around 2MB in size. Every time you make a change to your JS code (it can be a simple `console.log` removal), your client browser has to fetch the new version of that bundled JS file which is again around 2MB in size.

Code splitting is here to reduce the size of the file that needs to be fetched by separating your code from the “vendor” libraries. That way, if your code changes, the browser only has to fetch the file containing your code and not the file containing the libraries.

```
mix.js('resources/assets/js/app.js', 'public/js')
    .extract(['vue'])
```

The `extract` method accepts an array of all libraries or modules that you wish to extract into a `vendor.js` file.

Using the above command will generate the following files:

- `public/js/manifest.js` - The Webpack manifest runtime
- `public/js/vendor.js` - Your vendor libraries
- `public/js/app.js` - Your application code

To avoid JavaScript errors, be sure to load these files in the proper order:

```
<script src="/js/manifest.js"></script>
<script src="/js/vendor.js"></script>
<script src="/js/app.js"></script>
```

Copying

As with Laravel Elixir, Laravel Mix comes with copy task.

```
mix.copy('node_modules/foo/bar.css', 'public/css/bar.css');
```

You can copy files or directories. This is very helpful when you have to copy some fonts, CSS or JS files from `node_modules` directory to your `public` directory for further asset manipulation or reference.

Versioning

Another cool feature of Laravel Mix, but which was also available on Laravel Elixir is Versioning. Because files are cached in the browser for x period of time, there is no way of letting the browser know to fetch the latest version of the file if a change has occurred.

Versioning appends a unique hash at the end of the file, each time it changes so that the browser know to fetch the latest version.

```
mix.js('resources/assets/js/app.js', 'public/js')
    .version();
```

After running this task, you will not know the exact file name, so this is where Laravel's `mix` helper function comes into action. The `mix` function will automatically determine the correct file name:

```
<link rel="stylesheet" href="{{ mix('/css/app.css') }}">
```

Notifications

By default, OS notifications are displayed after each Laravel Mix task has completed. To disable the notifications you should add:

```
mix.disableNotifications();
```

to your `webpack.mix.js` file.

Thank you for reading this tutorial.

Sources

- [Compiling Assets \(Laravel Mix\)](#)⁹⁹
- [Compiling Assets \(Laravel Elixir\)](#)¹⁰⁰
- [Laravel Elixir Without Laravel](#)
- [Using Laravel Elixir in non Laravel project](#)
- [Laravel Elixir Will Be Renamed To Laravel Mix](#)¹⁰¹
- [Introducing Laravel Mix \(new in Laravel 5.4\)](#)¹⁰²

⁹⁹<https://laravel.com/docs/5.4/mix>

¹⁰⁰<https://laravel.com/docs/5.3/elixir>

¹⁰¹<https://laravel-news.com/laravel-elixir-to-laravel-mix>

¹⁰²<https://mattstauffer.co/blog/introducing-laravel-mix-new-in-laravel-5-4>

Laravel Mix Without Laravel

Even simpler asset manipulation than with Laravel Elixir. Highly recommended for beginners.

Published at: 16. February, 2017.



View Source Code

Source code for this tutorial is available [here](#)¹⁰³.

In my tutorial called [Laravel Elixir Without Laravel](#) we have gone over installing Laravel Elixir, configuring it to match our custom folder structure and adding tasks.

In this tutorial we will do the same thing, but with Laravel Mix. We will create a custom non-Laravel project, install Laravel Mix and make it handle our assets.

Installation

This is the current state of the repository at this moment: [5cc59c556b744a1091a9807a055e260dc40a5229](#)¹⁰⁴.

We will start with an empty folder. First, we have to create a file called `package.json` and inside it paste the following:

```
{
  "private": true,
  "scripts": {
    "dev": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=development node_modules/webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js",
    "watch": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=development node_modules/webpack/bin/webpack.js --watch --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js",
    "hot": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=development node_modules/webpack-dev-server/bin/webpack-dev-server.js --inline --hot --config=node_modules/laravel-mix/setup/webpack.config.js",
```

¹⁰³ <https://github.com/laravelista/laravel-mix-without-laravel>

¹⁰⁴ <https://github.com/laravelista/laravel-mix-without-laravel/commit/5cc59c556b744a1091a9807a055e260dc40a5229>


```

    "production": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=production node_modules/webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js"
  }
}

```

I have copied the code above from [package.json](https://github.com/laravel/laravel/blob/master/package.json)¹⁰⁵ which comes with the default Laravel 5.4 installation. It just sets the package to be private, meaning that you can't publish it by accident and it sets four scripts for running Laravel Mix. You can access them by using `npm run dev`, `npm run production`, `npm run watch` and `npm run hot`.

Now, we will install Laravel Mix. One of the requirements for this tutorial is to have [Node.js](https://nodejs.org/en/)¹⁰⁶ and NPM installed on your machine.

Run this command to install the dependencies and save them as development dependencies. *This means that these dependencies are only meant to be installed if you are working on changing the code (contributing), not when you are installing the package.*

```
npm install laravel-mix --save-dev
```

My `package.json` file now looks like this:

```

{
  "private": true,
  "scripts": {
    "dev": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=development node_modules/webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js",
    "watch": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=development node_modules/webpack/bin/webpack.js --watch --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js",
    "hot": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=development node_modules/webpack-dev-server/bin/webpack-dev-server.js --inline --hot --config=node_modules/laravel-mix/setup/webpack.config.js",
    "production": "node node_modules/cross-env/bin/cross-env.js NODE_ENV=production node_modules/webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-mix/setup/webpack.config.js"
  },
  "devDependencies": {
    "laravel-mix": "^0.8.1"
  }
}

```

¹⁰⁵<https://github.com/laravel/laravel/blob/master/package.json>

¹⁰⁶<https://nodejs.org/en/>

You can see that now you have a new directory called `node_modules`. This is where all dependencies that you install with NPM are held. Since we don't want that folder to be included in our repository we will add it to the `.gitignore` file. Create a `.gitignore` file in the root of the repository (next to the `package.json` file) and place the following inside:

```
/node_modules
```

Great, now the last thing to complete this installation. Create a file named `webpack.mix.js` in the root of the repository and paste this code inside:

```
const { mix } = require('laravel-mix');

//
```

We are now ready to proceed to the next step.

This is the current state of the repository at this moment: [5cc59c556b744a1091a9807a055e260dc40a5229](https://github.com/laravelista/laravel-mix-without-laravel/commit/5cc59c556b744a1091a9807a055e260dc40a5229)¹⁰⁷.

Folder structure

For this project, we will keep our assets in a `/src` directory and the compiled code in `/dist` directory. So, our structure will look like this:

```
/src
  /css
  /js
  /sass
  /less
/dist
  /css
  /js
```

For this project, we will use Sass for styles and JavaScript. Let's create some folders that we will need:

¹⁰⁷ <https://github.com/laravelista/laravel-mix-without-laravel/commit/5cc59c556b744a1091a9807a055e260dc40a5229>

```
mkdir src
mkdir src/sass
mkdir src/js
```

Adding tasks

Before we add the tasks for Sass and JavaScript, we will create files in appropriate directories with sample code.

Sass

Create a new file called `app.scss` in the `/src/sass` directory, and inside it paste the following:

```
body {
  background-color: red;
}
```

Nothing fancy, just to be able to check if everything works.

JavaScript

Again, nothing fancy here. Just a single line to test that everything works.

Create a new file called `app.js` in the `/src/js` directory, and inside it paste the following:

```
console.log('Fiddle me timbers!');
```

Awesome! We now have our first assets. Let's add tasks for them. Open your file `webpack.mix.js` and add this line at the end:

```
mix.sass('src/sass/app.scss', 'dist/css')
    .js('src/js/app.js', 'dist/js');
```

The first task is used to compile Sass into CSS. The `sass` method looks for `app.scss` in `src/sass` directory and puts the compiled CSS in `dist/css` directory.

The `js` method compiles ECMAScript 2015, bundles modules, does minification and concatenates plain JavaScript files. It looks for `app.js` in `src/js` directory and puts a single bundled file in the `dist/js` directory.

Let's give it a test run now. From the command line run:

```
npm run dev
```

You will get this output:

```
$ npm run dev
```

```
> @ dev C:\repos\laravel-mix-without-laravel
> node node_modules/cross-env/bin/cross-env.js NODE_ENV=development node_modules\
  webpack/bin/webpack.js --progress --hide-modules --config=node_modules/laravel-
  mix/setup/webpack.config.js
```

```
95% emitting DONE   Compiled successfully in 1510ms14:39:29
```

	Asset	Size	Chunks	Chunk Names
dist/js/app.js	2.78 kB	0 [emitted]	dist/js/app	
dist/css/app.css	486 bytes	0 [emitted]	dist/js/app	
mix-manifest.json	86 bytes	[emitted]		

This means that it all works.

Verification

To verify that it all works, check the following:

- Check if the `dist` directory with `css` and `js` directories inside it, are created
- Check that `dist/css/app.css` contains `background-color: red;` line
- Check that `dist/js/app.js` contains `console.log('Fiddle me timbers!');` (It should be near the end of the file).

If you could verify everything mentioned above, that means that adding the tasks was successful.

This is the current state of the repository at this moment: [7df60c59992f1d91b21c8cd30ce569922cd6ffe4](https://github.com/laravelista/laravel-mix-without-laravel/commit/7df60c59992f1d91b21c8cd30ce569922cd6ffe4)¹⁰⁸.

Usage

Run all tasks:

¹⁰⁸<https://github.com/laravelista/laravel-mix-without-laravel/commit/7df60c59992f1d91b21c8cd30ce569922cd6ffe4>

```
npm run dev
```

Run all tasks and minify:

```
npm run production
```

Run all tasks and watch for changes:

```
npm run watch
```

As you can see, it is much easier to do asset manipulation in a non-laravel project with Laravel Mix than it is with Laravel Elixir.

There is much more that you can do with Laravel Mix. Read all about it in the documentation [Compiling Assets \(Laravel Mix\)](#)¹⁰⁹ or in my tutorial [Sublimely Magnificent Laravel Mix](#).

Thank you for reading this tutorial.

¹⁰⁹<https://laravel.com/docs/5.4/mix>

Laravel Elixir Without Laravel

If you are tired of writing the same gulp tasks over and over for your projects, why not just install Laravel Elixir, and let it handle it all.

Published at: 23. January, 2017.



View Source Code

Source code for this tutorial is available [here](#)¹¹⁰.

I've written about this before in a post called [Using Laravel Elixir in non Laravel project](#). That was 2 years ago, and a lot has changed since. This time I will be using Laravel Elixir 6.0.0-15 and provide you with a working implementation of it in a custom project.

The reason why I am writing about this is because many times I've been in a situation where I needed to do asset management in a custom project of mine. Let's say that you are developing a plugin or working on a single page application, you will need to do asset management at some point. You will have to convert SASS or LESS to CSS, you will have to compile and minify your Javascript code, and other stuff.

So, why not just use something that you are used to using, like Laravel Elixir. I spend most of my time working on Laravel applications and Laravel Elixir is a great tool for asset management. Being able to use Laravel Elixir without a Laravel application is such a time saver.

Installation

This is the current state of the repository at this moment: [6f54bbe344fc2f5887542f286d4be2f2a3ea9b51](#)¹¹¹.

We will start with an empty folder. First, have to create a file called `package.json` and inside it paste the following:

¹¹⁰ <https://github.com/laravelista/laravel-elixir-without-laravel>

¹¹¹ <https://github.com/laravelista/laravel-elixir-without-laravel/commit/6f54bbe344fc2f5887542f286d4be2f2a3ea9b51>

```
{
  "private": true,
  "scripts": {
    "prod": "gulp --production",
    "dev": "gulp watch"
  }
}
```

I have copied the code above from [package.json](#)¹¹² which comes with the default Laravel 5.2 installation. It just sets the package to be private, meaning that you can't publish it by accident. Also, it sets two shortcuts for building for production and for continuous development. You can access them by using `npm run prod` or `npm run dev`.

Then, we will install Laravel Elixir, and Gulp which is required for Laravel Elixir to work. One of the requirements for this tutorial is to have [Node.js](#)¹¹³ installed on your machine. Run this command to install the dependencies and save them as development dependencies. *This means that these dependencies are only meant to be installed if you are working on changing the code (contributing), not when you are installing the package.*

```
npm install laravel-elixir gulp --save-dev
```

My `package.json` file now looks like this:

```
{
  "private": true,
  "scripts": {
    "prod": "gulp --production",
    "dev": "gulp watch"
  },
  "devDependencies": {
    "gulp": "^3.9.1",
    "laravel-elixir": "^6.0.0-15"
  }
}
```

You can see that now you have a new directory called `node_modules`. This is where all dependencies that you install with `npm` are held. Since we don't want that folder to be included in our repository we will add it to the `.gitignore` file. Create a `.gitignore` file in the root of the repository (next to `package.json`) and place the following inside:

¹¹²<https://github.com/laravel/laravel/blob/master/package.json>

¹¹³<https://nodejs.org/en/>

```
/node_modules
```

Great, now the last thing to complete this installation. Create a file named `gulpfile.js` in the root of the repository and paste this code inside:

```
const elixir = require('laravel-elixir');

elixir((mix) => {
  //
});
```

We are now ready to proceed to the next step.

This is the current state of the repository at this moment: [a4e24798c9ec4a82cf277848f68ee99291de0a7a](https://github.com/laravelista/laravel-elixir-without-laravel/commit/a4e24798c9ec4a82cf277848f68ee99291de0a7a)¹¹⁴.

Folder structure

By default, Laravel Elixir uses this folder structure:

```
/resources
  /assets
    /css
    /js
    /sass
    /less
    /stylus
/public
  /css
  /js
```

If you adopt this structure for your project, you don't have to change anything else. But, in most cases, you would want to change things a bit. For this project, we will keep our assets in a `/src` directory and the compiled code in `/dist` directory. So, our structure will look like this:

¹¹⁴<https://github.com/laravelista/laravel-elixir-without-laravel/commit/a4e24798c9ec4a82cf277848f68ee99291de0a7a>


```

/src
  /css
  /js
  /sass
  /less
  /stylus
/dist
  /css
  /js

```

For this project, we will use LESS for styles and Javascript. Let's create some folders that we will need:

```

mkdir src
mkdir src/less
mkdir src/js

```

Excellent. Now, we have to tell Laravel Elixir where to find our assets and where to place them once compiled. We do that by adding these two lines:

```

elixir.config.assetsPath = 'src';
elixir.config.publicPath = 'dist';

```

to `gulpfile.js` file, just below where we require Laravel Elixir `const elixir = require('laravel-elixir');`. Now our `gulpfile.js` looks like this:

```

const elixir = require('laravel-elixir');

elixir.config.assetsPath = 'src';
elixir.config.publicPath = 'dist';

elixir((mix) => {
  //
});

```

We have now configured Laravel Elixir to use our custom folder structure. **See how easy it was!**

Adding tasks

Before we add the tasks for LESS and JS, we will create files in appropriate directories with sample code.

LESS

Create a new file called `app.less` in `/src/less` directory, and inside it paste the following:

```
body {
  background-color: red;
}
```

Nothing fancy, just to be able to check if everything works.

JS

Again, nothing fancy here. Just a single line to test that everything works.

Create a new file called `app.js` in `/src/js` directory, and inside it paste the following:

```
console.log('Fiddle me timbers!');
```

Awesome! We now have our first assets. Let's add tasks for them. Open your file `gulpfile.js` and make the `elixir` function look like this:

```
elixir((mix) => {
  mix.less('app.less')
    .webpack('app.js');
});
```

The first task is used to compile LESS into CSS. The `less` method looks for `app.less` in `src/less` directory and puts the compiled CSS in `dist/css` directory.

The second task uses [Webpack](https://webpack.github.io/)¹¹⁵ module bundler. If you want you can use [Rollup](http://rollups.org/)¹¹⁶ instead. The `webpack` method compiles and bundles [ECMAScript 2015](https://babeljs.io/learn-es2015/)¹¹⁷ into plain JavaScript. It looks for `app.js` in `src/js` directory and puts a single bundled file in `dist/js` directory.

Let's give it a test run now. From the command line run:

```
gulp
```

You will get a message saying:

¹¹⁵<https://webpack.github.io/>

¹¹⁶<http://rollups.org/>

¹¹⁷<https://babeljs.io/learn-es2015/>

Installation Required

To use "mix.webpack()", please run the following command, and then trigger gulp \ again.

```
=====
npm install laravel-elixir-webpack-official --save-dev
=====
```

Let's do what the command says:

```
npm install laravel-elixir-webpack-official --save-dev
```

Now, rerun the command gulp. You will get this output:

Task	Summary	Source Files	Destination
mix.less()	1. Compiling Less	src\less\app.less	dist\css\app.css
	2. Autoprefixing CSS		
	3. Concatenating Files		
	4. Writing Source Maps		
	5. Saving to Destination		
mix.webpack()	1. Transforming ES2015 to ES5	src\js\app.js	dist\js\app.js
	2. Writing Source Maps		
	3. Saving to Destination		

This means that it all works.

Verification

To verify that it all works, check the following:

- Check if the `dist` directory with `css` and `js` directories inside it, are created
- Check that `dist/css/app.css` contains `background-color: red;` line
- Check that `dist/js/app.js` contains `console.log('Fiddle me timbers!');` (It should be near the end of the file).

If you could verify everything mentioned above, that means that adding the tasks was successful.

This is the current state of the repository at this moment: [58ad08744fe9a6cbaf98fb191199943cbf26f293](https://github.com/laravelista/laravel-elixir-without-laravel/commit/58ad08744fe9a6cbaf98fb191199943cbf26f293)¹¹⁸.

Usage

Run all tasks:

```
gulp
```

Run all tasks and minify all CSS and JavaScript:

```
gulp --production
```

or

```
npm run prod
```

Watching Assets For Changes:

```
gulp watch
```

or

```
npm run dev
```

There is much more that you can do with Laravel Elixir. Read all about it in the documentation [Compiling Assets \(Laravel Elixir\)](https://laravel.com/docs/5.3/elixir)¹¹⁹.

Thank you for reading this tutorial

¹¹⁸<https://github.com/laravelista/laravel-elixir-without-laravel/commit/58ad08744fe9a6cbaf98fb191199943cbf26f293>

¹¹⁹<https://laravel.com/docs/5.3/elixir>

Sources

- [Compiling Assets \(Laravel Elixir\)](#)¹²⁰
- [laravel-elixir NPM](#)¹²¹
- [laravel/laravel: A PHP Framework For Web Artisans](#)¹²²
- [laravel/elixir: Fluent API for Gulp](#)¹²³
- Using Laravel Elixir in non Laravel project

¹²⁰<https://laravel.com/docs/5.3/elixir>

¹²¹<https://www.npmjs.com/package/laravel-elixir>

¹²²<https://github.com/laravel/laravel>

¹²³<https://github.com/laravel/elixir>

General Topics

Learn how to create multilingual web application with Laravel. From simple text localization to database model translations.

Sitemap for better SEO

One fast and easy way to improve your standings with search engines is to provide a sitemap of all URLs that can be accessed on your website.

Published at: 12. March, 2016.



View Source Code

Source code for this tutorial is available [here](#)¹²⁴.

There is already a post on this website (you can find it in the archive) about using [Bard](#)¹²⁵ and [Laravel](#)¹²⁶ to create a sitemap. It contains some cool ways of adding dynamic links, multilingual links, named routes, sitemap index etc.

In this tutorial I will focus only on the core objective and that is a simple, plain sitemap with static URLs.

This kind of sitemap requires only a few minutes to set up, but it can increase your search engine standing a lot.

Package installation

For creating a sitemap we will be using a package that I have created called [Bard](#)¹²⁷.

To install Bard from your terminal type:

```
composer require laravelista/bard=~1.0.1
```

That is all you need.

Route and controller

Now let's create a SitemapController with a method called generate and a route for that method /sitemap.xml.

We need to create a controller using:

¹²⁴<https://github.com/laravelista/sitemap-for-better-seo>

¹²⁵<https://github.com/laravelista/Bard>

¹²⁶<https://laravel.com/>

¹²⁷<https://github.com/laravelista/Bard>

```
php artisan make:controller SitemapController
```

and create a constructor for that class that resolves our Sitemap dependency. Your whole controller will now look like this:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

use Laravelista\Bard\UrlSet as Sitemap;

class SitemapController extends Controller
{
    protected $sitemap;

    public function __construct(Sitemap $sitemap)
    {
        $this->sitemap = $sitemap;
    }
}
```

Take a look at this line `use Laravelista\Bard\UrlSet as Sitemap;`. Here I am assigning an alias for `UrlSet` class from `Bard` to `Sitemap`. *This is a handy shortcut to know.*

Let's create a new method called `generate` below the constructor and inside it call a method to render the sitemap:

```
public function generate()
{
    // future code goes here

    return $this->sitemap->render();
}
```

Now go to `app/Http/routes.php` and add a new route:


```
Route::get('sitemap.xml', 'SitemapController@generate');
```

If you open your browser on /sitemap.xml now and right-click -> view source code you will see:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9" xmlns:xhtml="http://\
www.w3.org/1999/xhtml"/>
```

This is the current state of the repository at this moment [11d3804610686030cd5f137022a30827d7a5ff9b](https://github.com/laravelista/sitemap-for-better-seo/tree/11d3804610686030cd5f137022a30827d7a5ff9b)¹²⁸.

Creating a sitemap

In the previous chapter, we have already created a sitemap but it is empty. Now we will add some URLs to it.

Above the render line in generate method place the following code.

If you are working on your own project be sure to use the correct URLs, and not the ones that are given here since they are just for demonstration purpose.

I will show you three different ways of adding URLs:

```
// first way
$home = $this->sitemap->addUrl(url('/'));
$home->setPriority(0.8);
$home->setChangeFrequency('hourly');
$home->setLastModification(\Carbon\Carbon::now());
$home->addTranslation("de", url('/de'));

// second way
$this->sitemap->addUrl(
    url('/contact'),
    0.5,
    null,
    new \DateTime('2015-04-01'),
    [
        "hreflang" => 'de',
        'href' => url('/de/contact')
```

¹²⁸<https://github.com/laravelista/sitemap-for-better-seo/tree/11d3804610686030cd5f137022a30827d7a5ff9b>

```
        ]  
    ];  
  
// third way  
$this->sitemap->addUrl(url('gallery'));
```

If you refresh your browser now, you will see that you have a sitemap with the URLs you defined.

This is the current state of the repository at this moment [9468b6c7de4a81925ed3444bfb0cf30fe52cb2a2](https://github.com/laravelista/sitemap-for-better-seo/tree/9468b6c7de4a81925ed3444bfb0cf30fe52cb2a2)¹²⁹.

This concludes this tutorial. I have shown you how to create a simple sitemap and as you can see it is very fast and easy.

¹²⁹ <https://github.com/laravelista/sitemap-for-better-seo/tree/9468b6c7de4a81925ed3444bfb0cf30fe52cb2a2>

RSS feed for news readers

So many people these days are using news readers to stay up to date with the latest content and news. Having an RSS feed for your website provides another way for your users to be notified of fresh content or news.

Published at: 12. March, 2016.



View Source Code

Source code for this tutorial is available [here](#)¹³⁰.

If you haven't already got a news reader check out [Feedly](#)¹³¹. There are many others out there but this is the one I personally use. I am subscribed to a lot of sites and blogs like [Laracasts](#)¹³², [Codeception](#)¹³³, [murze.be](#)¹³⁴, [Laravel Podcasts](#)¹³⁵, [Culttt](#)¹³⁶, [Laravel News](#)¹³⁷ etc.

Having a way for your users to be able to subscribe to receive the latest news on their news readers is awesome and increases chances that that user will come back to your site.

Who knows, I may even subscribe to your website ;)

Project preparation

If you are working on your own project and you already have a database setup with a table/model that you want to display on your RSS feed, you can skip this part.

I have created a blank laravel project with one table that holds *posts* with a model called `Post`. You can view the source code on GitHub. Also be sure to read the `readme.md` file that comes with it. You need to create a database file, migrate the database and seed the table with fake posts before continuing.

This is the current state of the repository at this moment [2543dfc3aa9e6e6e32be118560fd05dc4e923494](#)¹³⁸.

¹³⁰<https://github.com/laravelista/rss-feed-for-news-readers>

¹³¹<https://feedly.com/>

¹³²<https://laracasts.com/>

¹³³<http://codeception.com/>

¹³⁴<https://murze.be/>

¹³⁵<http://www.laravelpodcast.com/>

¹³⁶<http://culttt.com/>

¹³⁷<https://laravel-news.com/>

¹³⁸<https://github.com/laravelista/rss-feed-for-news-readers/tree/2543dfc3aa9e6e6e32be118560fd05dc4e923494>

Package installation

We will be using [roumen/feed](#)¹³⁹ package for generating the feed.

To install it from you terminal type:

```
composer require roumen/feed=~2.10
```

Then register this service provider with Laravel in config/app.php under *providers*:

```
Roumen\Food\FoodServiceProvider::class,
```

and add class alias in the same file under *aliases*:

```
'Feed' => Roumen\Food\Food::class,
```

Route and controller

Now we will create a controller for our feed called FeedController. From your terminal type:

```
php artisan make:controller FeedController
```

And inside it, create a method called generate so that it looks like this:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

class FeedController extends Controller
{
    public function generate()
    {

    }
}
```

We need to create a route for that method, so let's go to app/Http/routes.php file and create a route:

¹³⁹<https://github.com/RoumenDamianoff/laravel-feed>

```
Route::get('feed', 'FeedController@generate');
```

Now if you point your browser to /feed you should get a blank page.

This is the current state of the repository at this moment [d83a9f98338a8ec0c58f049cb453f92095adb41d](https://github.com/laravelista/rss-feed-for-news-readers/tree/d83a9f98338a8ec0c58f049cb453f92095adb41d)¹⁴⁰.

Generating feed

Now that all the preparation are complete we are free to get started creating the feed and populating it with content.

First we need to fetch latest 10 published posts from our database. We do that by placing this line in generate method:

```
$posts = \App\Post::latest()->published()->take(10)->get();
```

Now let's create the feed. Add this code and modify it to your liking:

```
$feed = \App::make('feed');
$feed->title = 'My blog';
$feed->description = 'This is my blog about this';
// $feed->logo = asset('img/logo.png'); //optional
$feed->link = url('feed');
$feed->setDateFormat('datetime'); // 'datetime', 'timestamp' or 'carbon'
$feed->pubdate = $posts[0]->created_at;
$feed->lang = 'en';
$feed->setShortening(true); // true or false
$feed->setTextLimit(100); // maximum length of description text
```

We still have to populate the feed with our posts and render it:

¹⁴⁰ <https://github.com/laravelista/rss-feed-for-news-readers/tree/d83a9f98338a8ec0c58f049cb453f92095adb41d>

```
foreach ($posts as $post)
{
    // set item's title, author, url, pubdate, description and content
    $feed->add($post->title, 'Author', url('posts/' . $post->id), $post->created\
_at, $post->content, $post->content);
}

return $feed->render('rss'); // or atom
```

Now if you reload the page you should get a valid RSS feed. One last thing to do is to add a link to your feed from your website.

Add this code between <head> tag in your layout file:

```
<link rel="alternate" type="application/rss+xml" title="My blog" href="{{ url('f\
eed') }}" />
```

This way the news readers can detect that your website has a feed.

This is the current state of the repository at this moment [53f4fee743dd5f023745b9c9275a6b28286453fc](https://github.com/laravelista/rss-feed-for-news-readers/tree/53f4fee743dd5f023745b9c9275a6b28286453fc)¹⁴¹.

And that's it. You now have a working RSS feed for your website.

¹⁴¹<https://github.com/laravelista/rss-feed-for-news-readers/tree/53f4fee743dd5f023745b9c9275a6b28286453fc>

Newsletter subscription with MailChimp

In this tutorial, I will show you how to set up a simple form for collecting email addresses for your newsletter using MailChimp.

Published at: 12. March, 2016.



View Source Code

Source code for this tutorial is available [here](#)¹⁴².

So imagine that you want to subscribe your website visitors to your newsletter or you are starting a brand new product/service and you want to collect email address of anyone that is interested so that you can send them an email when it is published or any other scenario where you would want to collect visitor emails for a purpose of later sending them an email.

In this tutorial, I will show you how to create a free [MailChimp](#)¹⁴³ account, create a list for newsletter subscribers and use Laravel to create a subscribe form for that list.

You can view the whole [source code](#)¹⁴⁴ for this tutorial on GitHub.

MailChimp account

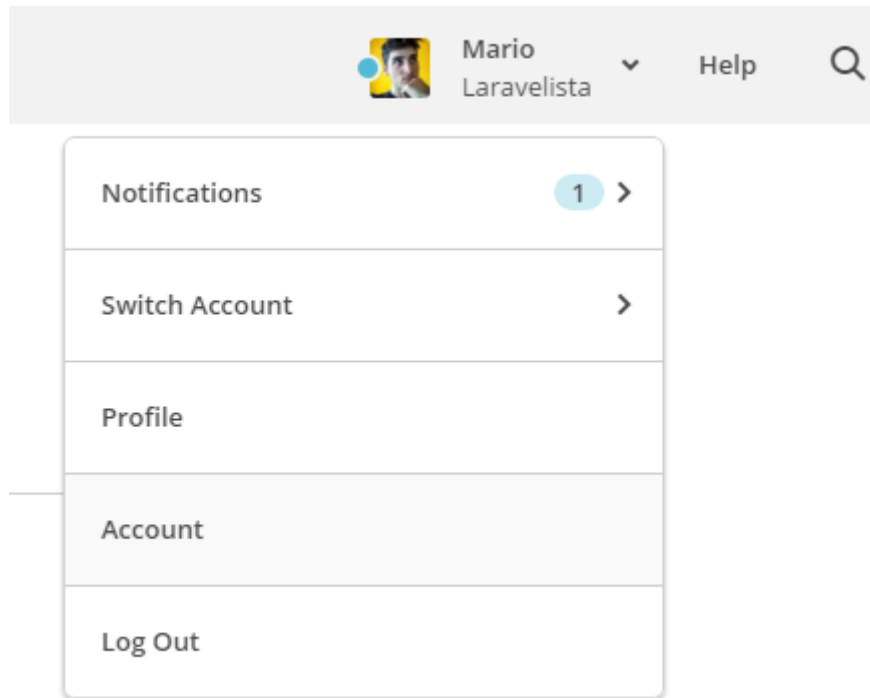
Let us start by creating an account on [MailChimp](#)¹⁴⁵. Once logged in, click on your Name in the upper right corner and go to *Account*. There under *Your API keys* click on *Create a key* and once completed write down the API key. **You will need that API key later.**

¹⁴²<https://github.com/laravelista/newsletter-subscription-with-mailchimp>

¹⁴³<http://mailchimp.com/>

¹⁴⁴<https://github.com/laravelista/newsletter-subscription-with-mailchimp>

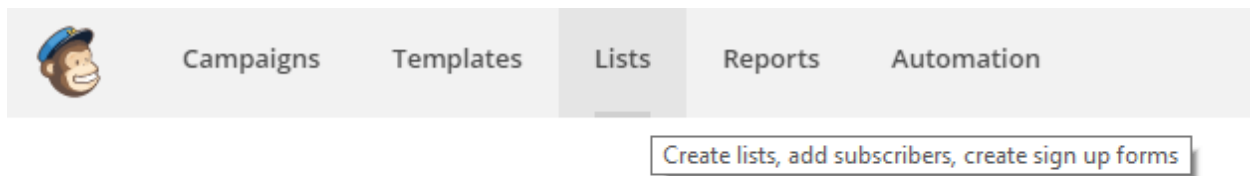
¹⁴⁵<http://mailchimp.com/>



Account

Newsletter list

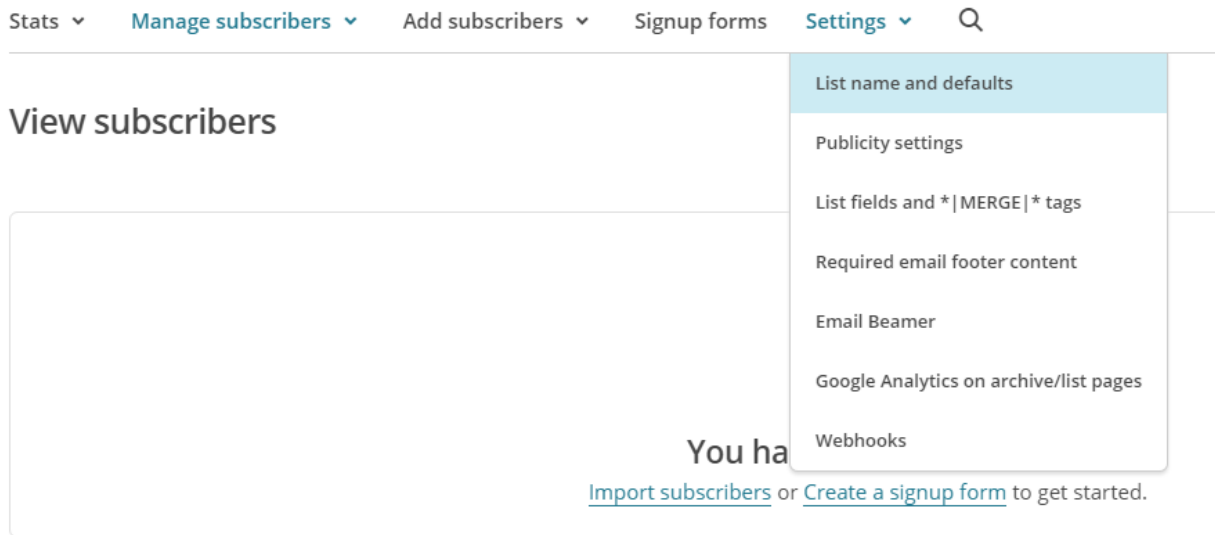
Now, let's create a list that will contain all subscribers to the newsletter. Click *Lists* on the menu and then click on *Create List* button.



Lists

Lists

Set the *List name* to newsletter and fill in any required fields. Once created go to *Settings* -> *List name and defaults*. There you will find *List ID*. Write that down next to the API key that you have recently written down. **You will need this List ID later.**



List name and defaults

List ID

Some plugins and integrations may request your List ID.

Typically, this is what they want: **181710f645**.

List ID

This part is complete, you are free to look around MailChimp and tinker with options as you please. The next part of this tutorial is done in Laravel.

Laravel and MailChimp package

There are many ways of getting this done, but in this tutorial, I will use the simplest way while still doing things the Laravel way.

I have created a repository for this tutorial on [GitHub](https://github.com/laravelista/newsletter-subscription-with-mailchimp)¹⁴⁶ so you can follow along.

This is the current state of the repository at this moment [aaf6895106a76353387d2b5fab86ba833647014b](https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/aaf6895106a76353387d2b5fab86ba833647014b)¹⁴⁷.

I am working with Laravel Framework version 5.2.22

¹⁴⁶ <https://github.com/laravelista/newsletter-subscription-with-mailchimp>

¹⁴⁷ <https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/aaf6895106a76353387d2b5fab86ba833647014b>

Package installation

Let's start by installing the [mailchimp package](https://packagist.org/packages/mailchimp/mailchimp)¹⁴⁸ using Composer.

Type in terminal:

```
composer require mailchimp/mailchimp=~2.0
```

And that's it. You have now installed the MailChimp package in your Laravel application.

Setting up a simple form

We will now create a simple form using [Bootstrap](http://getbootstrap.com/)¹⁴⁹ to collect users email addresses.

Replace all content in file `resources/views/welcome.blade.php` with:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Newsletter subscription</title>

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap.min.css" integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDD\
djZlpLegxhjVME1fgjWPGmkzs7" crossorigin="anonymous">
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-md-4 col-md-offset-4">
          <h1 class="text-center">Subscribe to our newsletter!</h1>

          <form action="{{ url('subscribe') }}" method="POST">
            {!! csrf_field() !!}
            <div class="form-group">
              <label for="email">Email</label>
              <input type="email" class="form-control" placeholder="Em\
ail" name="email">
            </div>
          </form>
        </div>
      </div>
    </div>
  </body>
</html>
```

¹⁴⁸<https://packagist.org/packages/mailchimp/mailchimp>

¹⁴⁹<http://getbootstrap.com/>

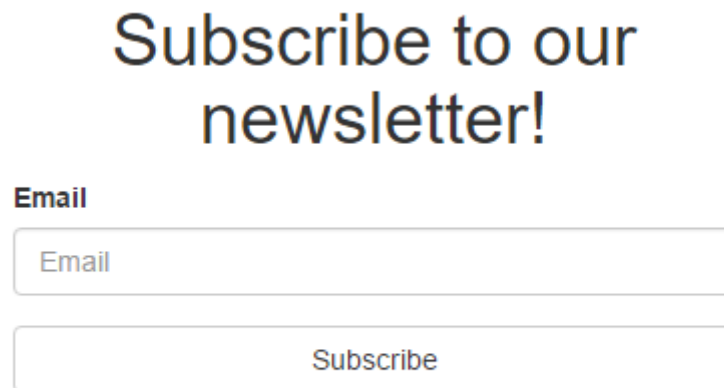
```

        </div>
        <button type="submit" class="btn btn-default btn-block">Subs\
cribe</button>
    </form>
</div>
</div>
</div>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.\
js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.mi\
n.js" integrity="sha384-0mSbJDEHialfmuBBQP6A4Qrprq50Vfw37PRR3j5ELqxs1yVq0tnepnH\
VP9aJ7xS" crossorigin="anonymous"></script>
</body>
</html>

```

The code above adds Bootstrap to our page and provides a nice little form for collecting email addresses which points to /subscribe URL on our application.



Newsletter form

This is the current state of the repository at this moment [129ca100e39dc1bb64e1c348a9d9a3ae553dd0a1](https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/129ca100e39dc1bb64e1c348a9d9a3ae553dd0a1)¹⁵⁰.

TIP: You can run the application using `php artisan serve` and open your browser on `http://localhost:8000` if you are following along with GitHub.

¹⁵⁰ <https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/129ca100e39dc1bb64e1c348a9d9a3ae553dd0a1>

Route, controller, validation and action

Now that we have our form how we want it, we can proceed to setup our route that will handle the form action.

This could all be easily done within a route closure, but let's keep it nice and clean instead. We will create a newsletter controller that will handle subscription, a route for that method, apply validation and finally add the subscriber to the list that we have previously created on MailChimp.

Let's start by creating the newsletter controller. In your terminal write:

```
php artisan make:controller NewsletterController
```

Open that file `app/Http/Controllers/NewsletterController` and add a method called `subscribe` so that your file looks like this:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;

use App\Http\Requests;

class NewsletterController extends Controller
{
    public function subscribe()
    {

    }
}
```

Now let's create a route for that method. Open routes file `app/Http/routes.php` and add this line to *Application Routes* group.

```
Route::post('subscribe', 'NewsletterController@subscribe');
```

Also because we want to have **CSRF protection** we will have to move our default / route to the *Application Routes* group so that it looks like this:

```
Route::group(['middleware' => ['web']], function () {
    Route::get('/', function () {
        return view('welcome');
    });
    Route::post('subscribe', 'NewsletterController@subscribe');
});
```

Now if you refresh your page and press on *Subscribe* you should get an empty page.

Before we continue, we will add validation to our subscribe method so that the *Email* field is required and that it is in valid email format.

To the NewsletterController subscribe method add a dependency Request \$request:

```
public function subscribe(Request $request)
```

and add this code inside it:

```
$this->validate($request, [
    'email' => 'required|email'
]);
```

In order to be able to see if there are any validation errors we have to add this code above our newsletter subscription form and just below h1 tag:

```
@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

Subscribe to our newsletter!

- The email field is required.

Email

Form validation

This is the current state of the repository at this moment [4e56219340a9d5145daa0d556d294cc097482c18](https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/4e56219340a9d5145daa0d556d294cc097482c18)¹⁵¹.

Now that all preparations are complete we can move to the fun part.

Remember the keys from MailChimp from the start of this tutorial that I've told you to write down somewhere? Well, now you will need those.

Open file `config/services.php` and just below stripe add a new key for mailchimp like so:

```
'mailchimp' => [  
    'api_key' => env('MAILCHIMP_API_KEY')  
]
```

Now open your `.env` file and add `MAILCHIMP_API_KEY=your-api-key` (Replace `your-api-key` with your real API key there).

Go back to `NewsletterController` and add a constructor and a property for MailChimp there like so:

¹⁵¹<https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/4e56219340a9d5145daa0d556d294cc097482c18>

```
protected $mailchimp;

public function __construct()
{
    $this->mailchimp = new \MailChimp(config('services.mailchimp.api_key'));
}
```

In our subscribe method now we will add code to add the entered email address to our MailChimp newsletter list.

Add this below the validation in subscribe method:

```
$listId = '181710f645'; // Replace this with your own List ID

try {
    $this->mailchimp->lists->subscribe($listId, $request->only(['email']));
} catch (\MailChimp_Error $e) {
    if ($e->getMessage()) {
        $request->session()->flash('errors',
            $this->createViewError('mailchimp_error', $e->getMessage()));
        return redirect()->back();
    } else {
        $request->session()->flash('errors',
            $this->createViewError('mailchimp_error', 'An unknown error occurred\
'));
        return redirect()->back();
    }
}
```

And this is the code for createViewError protected method:

```
protected function createViewError($key, $value)
{
    return collect([$key => $value]);
}
```

This method enables us to display MailChimp errors above the form.

Upon verifying the code above I have been greeted with the following error

```
API call to lists/subscribe failed: SSL certificate problem: unable to get local\
issuer certificate
```

This has to do something with cURL and the way I have managed to solve is to download `ca-cert.pem`¹⁵² file to my PC (C:/tools/ca-cert.pem) and tell PHP where to find it by adding this code to my `php.ini`:

```
[curl]
; A default value for the CURLOPT_CAINFO option. This is required to be an
; absolute path.
curl.cainfo = "C:/tools/ca-cert.pem"
```

Now when you test the form with a valid email address you should get a blank page and if there is an error you will see it above the form in the red alert box.

This is the current state of the repository at this moment [9f0e21e6d1c286eea28e7b589585dd9dbd7ca174](https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/9f0e21e6d1c286eea28e7b589585dd9dbd7ca174)¹⁵³.

In the next chapter, we will create a simple “Thank you for subscribing page” for our subscribers.

User feedback

So far, our newsletter form works, but once the user has submitted the form it shows a blank page. We will change that by redirecting the user to our “Thank you for subscribing page” if there are no errors.

Let’s start by creating a view. Create a new file `resources/views/thankyou.blade.php` and paste this inside it:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Thank you for subscribing to our newsletter</title>

    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6\
/css/bootstrap.min.css" integrity="sha384-1q8mTJ0ASx8j1Au+a5WDVnPi2lkFfwwEAa8hDD\
djZlpLegxhjVME1fgjWPGmkzs7" crossorigin="anonymous">
  </head>
  <body>
    <div class="container">
      <div class="row">
        <div class="col-md-12">
```

¹⁵² <https://curl.haxx.se/ca/ca-cert.pem>

¹⁵³ <https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/9f0e21e6d1c286eea28e7b589585dd9dbd7ca174>


```

        <br />
        <div class="jumbotron">
            <h1>Success!</h1>
            <p>Thank you for subscribing to our newsletter.</p>
        </div>
    </div>
</div>

<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.\
js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.mi\
n.js" integrity="sha384-0mSbJDEHialfmuBBQP6A4Qrprq50Vfw37PRR3j5ELqxs51yVq0tnepnH\
VP9aJ7xS" crossorigin="anonymous"></script>
</body>
</html>

```

Now we need to add a route for that view. Go to `app/Http/routes.php` and in *Application Routes* group append a new route:

```

Route::get('thankyou', function() {
    return view('thankyou');
});

```

You can test that the route and the view work by pointing your browser to `/thankyou` URL.

One last step to make this work is to add a redirect line to our subscribe method.

In `NewsletterController` just below the try-catch statement add a line:

```

return redirect('/thankyou');

```

This line redirects the user to our newly created view if the try-catch statement passes.

Success!

Thank you for subscribing to our newsletter.

Thank you for subscribing page

This is the current state of the repository at this moment [30d8222b8b754a3fd9a913fea803091fef3494e3](https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/30d8222b8b754a3fd9a913fea803091fef3494e3)¹⁵⁴.

You now have a working newsletter subscription form with MailChimp. *Be sure to check out the repository on GitHub if unclear on something.*

¹⁵⁴<https://github.com/laravelista/newsletter-subscription-with-mailchimp/tree/30d8222b8b754a3fd9a913fea803091fef3494e3>

PHPUnit Code Coverage Analysis

In this tutorial, I will demonstrate on my existing Laravel package called Ekko how to configure code coverage on PHPUnit, analyze given results and improve tests to achieve maximum code coverage.

Published at: 29. June, 2016.



View Source Code

Source code for this tutorial is available [here](#)¹⁵⁵.

Code coverage helps you detect which areas of your application are not covered with tests. Imagine if you have a number of classes in your domain and you have tests for them. In order to determine which method has tests written for it, you would have to manually check the tests to see if it has any tests that cover that method. Also, you would have to check if the test for that method covers all possibilities inside that method; This refers to if statements and branching.

As it turns out, a package I wrote called [Ekko](#)¹⁵⁶ that helps you detect which menu item is active based on current URL and apply an appropriate CSS class to that item, already has tests written for it in PHPUnit. This is the perfect scenario for demonstrating what code coverage is, how to achieve maximum code coverage and why.

Xdebug installation

To be able to run code coverage on PHPUnit, you will have to have Xdebug installed and configured on your system. These are some of your options:

- If you are using **Homestead**, you can ssh into the machine and run code coverage from there since it already comes with Xdebug installed
- If you are using **Laragon**, it also already comes with Xdebug, so you can use its shell to run code coverage
- If you are on **Windows**, you will have to install the correct version of Xdebug that matches your PHP version

¹⁵⁵<https://github.com/laravelista/Ekko>

¹⁵⁶<https://github.com/laravelista/Ekko>

- For Mac and Linux installation instructions see the [documentation](#)¹⁵⁷

The most painless way of getting started with Xdebug is either with Homestead or Laragon. If you can, use those.

Ahoy, Windows users over here

If you are like me, then you will want to have Xdebug installed on your host machine, same as PHP mentioned in [Laravel on Windows with Homestead](#) course.

The procedure is really simple. You just have to know which version of PHP you have installed on your system, download the corresponding version of Xdebug, place it in the extensions directory and modify your `php.ini` file. *Piece of cake!*

There are a few pointers that I can give you:

- Check your PHP version with `php -v` from the command line. Download Xdebug binaries that match your PHP version
- If you downloaded PHP version marked with “Thread Safe aka TS”, you have to download Xdebug binary marked with “TS”
- If you don’t see “nts” in your PHP version folder name, then you have “TS” version
- The last thing to notice is the architecture x64 or x86. If you are running a 64bit operating system as you should, you should check if your PHP version is x86 or x64, because it can be both on 64bit systems. Xdebug must be for the same architecture as your PHP.

For example. If I download [PHP zip file](#)¹⁵⁸ named `php-7.0.8-Win32-VC14-x64.zip`, I should [download Xdebug](#)¹⁵⁹ for PHP 7.0 VC14 TS (64bit).

Once you have downloaded the appropriate version, move it to inside your PHP directory in `ext` directory. Now go one folder up and open `php.ini` file using your favorite text editor. At the end of the file add this line:

```
zend_extension = ext\php_xdebug-2.4.0-7.0-vc14-x86_64.dll
```

If you run `php -v` now from the command line, you should get an output something like this, depending on your version:

¹⁵⁷<https://xdebug.org/docs/install>

¹⁵⁸<http://windows.php.net/download/>

¹⁵⁹<https://xdebug.org/download.php>

```
PHP 7.0.0 (cli) (built: Dec 3 2015 11:36:59) ( ZTS )
Copyright (c) 1997-2015 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2015 Zend Technologies
    with Xdebug v2.4.0, Copyright (c) 2002-2016, by Derick Rethans
```

If you get a warning of some kind regarding Xdebug at this point, you messed something up :) Retrace your steps and try again. If you are still having problems, let me know in the comments below and I will do my best to help you.

First code coverage report

So that you can follow along, if you want, you need to clone Ekko repository from GitHub and checkout commit 8355f9efdba816443a39d5fd3cf396fec30b26d3 using:

```
git clone git@github.com:laravelista/Ekko.git
```

```
cd Ekko
```

```
git checkout 8355f9efdba816443a39d5fd3cf396fec30b26d3
```

This will place you in the code before code coverage was implemented so that you can see the changes as I progress in the tutorial.

Before we can run PHPUnit with code coverage we need to define a *whitelist* telling PHPUnit which files to include in the report. To do this we need to add a section to `phpunit.xml` file located in the root of the repository. Open `phpunit.xml` and add:

```
<filter>
    <whitelist processUncoveredFilesFromWhitelist="true">
        <directory suffix=".php">./src</directory>
    </whitelist>
</filter>
```

below the closing `</testsuites>` tag. This snippet above will include all `.php` files located in `/src` directory.

Now from the command line, we run the command:

```
phpunit --coverage-html ./coverage
```

The last parameter tells PHPUnit where to store the report.

This is the output from the command above:

```
$ phpunit --coverage-html ./coverage
```

```
PHPUnit 5.3.2 by Sebastian Bergmann and contributors.
```

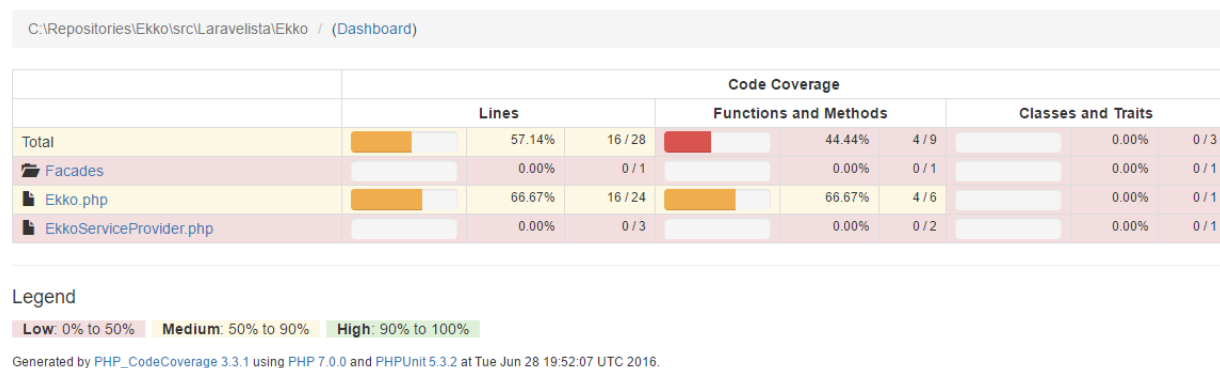
```
...
```

```
Time: 3.58 seconds, Memory: 6.00Mb
```

```
OK (3 tests, 16 assertions)
```

```
Generating code coverage report in HTML format ... done
```

To view the report navigate to `./coverage` directory and open in browser file `index.html`. You should get a report like this:



First code coverage report

Configure report to match your needs

As you can see in the first row, a total number of lines covered is 16/28 and 4 out of 9 methods. First thing to notice here is that in the report is included the Facades folder and EkkoServiceProvider.php file.

Since we are only interested in covering main Ekko class located in Ekko.php file we need to exclude all other files and folders.

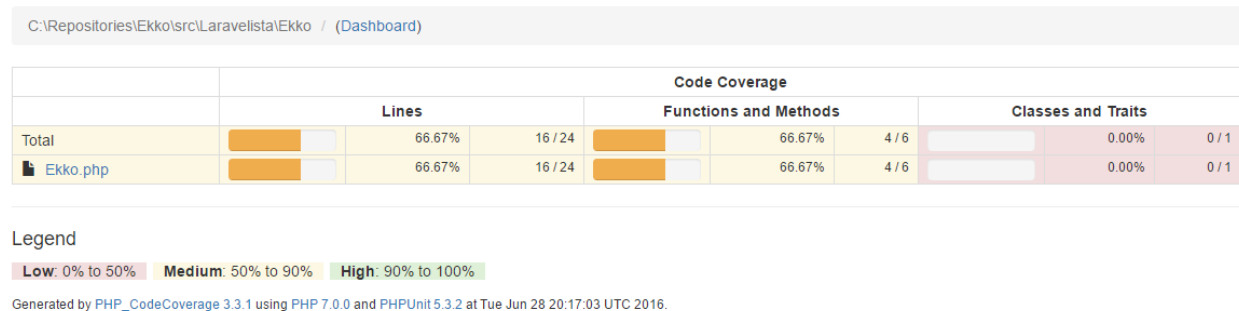
Inside the `<whitelist>` tag, just bellow the `<directory>` tag, add this:

```
<exclude>
  <directory suffix=".php"> ./Laravelista/Ekko/Facades</directory>
  <file> ./Laravelista/Ekko/EkkoServiceProvider.php</file>
</exclude>
```

Now run the same command again to generate a new report:

```
phpunit --coverage-html ./coverage
```

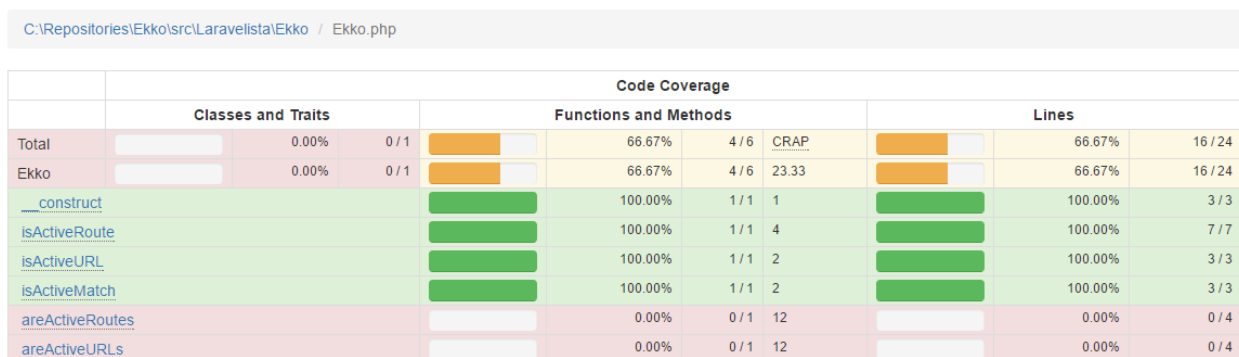
This time the report looks more relevant to us:



Relevant report

Code coverage analysis

If you click on Ekko.php you will see the whole class marked with red and green. The green lines are covered with tests, while red lines are not.



Ekko coverage

As you can see methods `areActiveRoutes` and `areActiveURLs` have zero code coverage. This short and simple report tells us that we need to write tests for those two methods in order to achieve maximum code coverage, which is what we aim at if possible.

To view the code at this point you need to checkout commit `86e339c0234e72951a69a3d37c44bd7df2ffaa3c` using:

```
git checkout 86e339c0234e72951a69a3d37c44bd7df2ffaa3c
```

Improving code coverage

Now a little bit of theory and thinking. There is a reason why I initially did not write tests for these two methods. If you look at those two methods you will see that they just wrap `isActiveURL` or `isActiveRoute` in for each loop for which we already have tests. Because these are very simple methods, there is no benefit in writing tests for them. The important methods are all tested, that is what is important.

Now, if you want to achieve maximum code coverage you will have to write tests for those two methods also. This is a very small example, but at this point, you should ask yourself “is it worth it?”. Is achieving maximum code coverage your primary goal while testing or is it to test the important stuff. The trade off is like this. You can spend time writing tests for those two methods to get 100% code coverage or you can spend that time on something else, maybe more important.

My suggestion is this: If you are on a deadline, you can skip tests for those methods or come back to them later. Focus your attention on what matters most. Your goal while testing is to achieve 100% code coverage, but don't sacrifice your time to get there. Test what is of critical importance and leave the rest for edge cases. Or if you are a perfectionist, just test everything but realize that you will spend a lot of time writing tests. That time you could have spent on something more productive like drinking a cup of tea.

The temptation to achieve maximum code coverage is huge and I myself have fallen into that trap many times. **You, as a developer have a natural need to achieve 100% of everything.** Sometimes, it's just not worth it. It is up to you to decide.

Now to achieve that 100% code coverage haha.

Maximum code coverage

I have added tests for the two mentioned methods and now our report looks like this:

C:\Repositories\Ekko\src\Laravelista\Ekko / Ekko.php

	Code Coverage									
	Classes and Traits			Functions and Methods				Lines		
Total	<div></div>	100.00%	1 / 1	<div></div>	100.00%	6 / 6	CRAP	<div></div>	100.00%	24 / 24
Ekko	<div></div>	100.00%	1 / 1	<div></div>	100.00%	6 / 6	15	<div></div>	100.00%	24 / 24
__construct	<div></div>			<div></div>	100.00%	1 / 1	1	<div></div>	100.00%	3 / 3
isActiveRoute	<div></div>			<div></div>	100.00%	1 / 1	4	<div></div>	100.00%	7 / 7
isActiveURL	<div></div>			<div></div>	100.00%	1 / 1	2	<div></div>	100.00%	3 / 3
isActiveMatch	<div></div>			<div></div>	100.00%	1 / 1	2	<div></div>	100.00%	3 / 3
areActiveRoutes	<div></div>			<div></div>	100.00%	1 / 1	3	<div></div>	100.00%	4 / 4
areActiveURLs	<div></div>			<div></div>	100.00%	1 / 1	3	<div></div>	100.00%	4 / 4

Maximum code coverage

To view the code at this point you need to checkout commit `7a4aaca0d46c4ace4f63b7212c88002b3ba3bcf8` using:


```
git checkout 7a4aaca0d46c4ace4f63b7212c88002b3ba3bcf8
```

I feel pretty happy about this now.

Conclusion

The benefit to adding tests for those two methods is almost close to zero. The package is very small and I have done this mostly to demonstrate how you can review your code using code coverage. Since Laravel comes with PHPUnit baked in, you can apply the same steps taken in this tutorial to code coverage your Laravel application or any kind of PHP package. Viewing code coverage reports helps you spot poorly tested parts of your application or package.

I am more of a visual type myself and I like seeing the report because I feel the sense of progress and that drives me to go further.

This concludes this tutorial.

Creating a Sitemap with Bard 2.0

This version of Bard brings simpler (new) syntax, easier sitemap creation, and it will completely break your application if you have used the older version of Bard before :)

Published at: 05. July, 2017.

A few days ago, I published a website [Visit Murter](https://visitmurter.com)¹⁶⁰ where you can explore and see the destinations on the island Murter. It is a relatively big website and I needed to create a sitemap for it. I already had a package for sitemap creation called [Bard](https://github.com/laravelista/Bard)¹⁶¹, but at that moment the last commit to Bard was made in March 2015.

The package works and I have used it on many sites that I have created before. Even this site (Laravelista) uses Bard for Sitemap creation, but after working with it many times I have come to a conclusion that I can make it easier to get started with Bard.

I must admit, that every time that I did `composer require laravelista/bard` my next step was going to the documentation and figuring out how to implement the “Laravel boilerplate” that came with Bard. The boilerplate enabled you to easily implement translations and use route names instead of URLs. It was something that I was proud when I first created it, but soon I realized that it was a big pain in the ass.

This new version of Bard almost completely breaks the old way of doing things. Before you could have used the constructor to create the URL or you could have set only the location and then use multiple method calls to set the desired properties on the URL. In this version, there is only one way of doing things and that is by chaining methods. Also, setting translations has also been simplified.

In this tutorial, I will show you how to write a simple PHP file that generates an example sitemap using Bard, and in the next tutorial, I will show you how to use Bard and Laravel on the same example.

New project

In the place where you keep your repositories aka projects, create a new folder and call it `bard-php`. Inside it run `composer init`. Follow the setup process and at the end confirm generation.

This is what I got:

¹⁶⁰<https://visitmurter.com>

¹⁶¹<https://github.com/laravelista/Bard>

```
{
  "name": "laravelista/bard-php",
  "description": "Example PHP script that demonstrates how to create a sitemap\
with Bard.",
  "license": "MIT",
  "authors": [
    {
      "name": "Mario Bašić",
      "email": "mario.basic@outlook.com"
    }
  ],
  "require": {}
}
```

Now to create a license file. Create a file called `license.md` and inside it paste the following:

The MIT License (MIT)

Copyright (c) 2017 Mario Bašić

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Replace the Author name and year with the one that suits you.

I will create a readme file for this project `readme.md`. It will be pushed to GitHub and that is why we are creating the license and readme files.

This is the current state of the repository at this moment [b10660acf7982d3a2c1d39e0d630f15e46069949](https://github.com/laravelista/creating-a-sitemap-with-bard-2-0/commit/b10660acf7982d3a2c1d39e0d630f15e46069949)¹⁶².

Installation

Now to install Bard we need to run:

```
composer require laravelista/bard
```

Output:

```
$ composer require laravelista/bard
Using version ^2.0 for laravelista/bard
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing symfony/polyfill-mbstring (v1.4.0)
  Downloading: 100%

- Installing symfony/translation (v3.3.3)
  Downloading: 100%

- Installing nesbot/carbon (1.22.1)
  Loading from cache

- Installing symfony/http-foundation (v3.3.3)
  Downloading: 100%

- Installing sabre/uri (1.2.1)
  Loading from cache

- Installing sabre/xml (0.4.3)
  Loading from cache

- Installing laravelista/bard (2.0.0)
  Loading from cache
```

This means that Bard is installed, but there are a few things that we need to do now. We need to tell Git to ignore the vendor folder. We do that by creating a `.gitignore` file in the root of our project and inside it write `vendor/` like so:

.gitignore

¹⁶²<https://github.com/laravelista/creating-a-sitemap-with-bard-2-0/commit/b10660acf7982d3a2c1d39e0d630f15e46069949>

vendor/

Awesome!

This is the current state of the repository at this moment [77b94ee58ff68e9c3564ad4f9c621430187f46fb](https://github.com/laravelista/creating-a-sitemap-with-bard-2-0/commit/77b94ee58ff68e9c3564ad4f9c621430187f46fb)¹⁶³.

Creating a sitemap

Create a file called `sitemap.php` and inside it place the following code:

```
<?php
```

```
require('vendor/autoload.php');
```

```
echo 'hello';
```

Now run `php -S localhost:8000` to test that everything is working. Point your browser to `http://localhost:8000/sitemap.php`. You should see hello text displayed.

Great!

Now let's replace the line `echo 'hello';` with the following:

```
use Laravelista\Bard\UrlSet;
```

```
use Sabre\Xml\Writer;
```

```
use Carbon\Carbon;
```

```
$sitemap = new UrlSet(new Writer);
```

```
$sitemap->addUrl('http://domain.com')
    ->setPriority(1.0)
    ->setChangeFrequency('always')
    ->addTranslation('hr', 'http://domain.com/hr');
```

```
$sitemap->addUrl('http://domain.com/contact')
    ->setPriority(0.8)
    ->setChangeFrequency('hourly')
    ->setLastModification(Carbon::now())
    ->addTranslation('hr', 'http://domain.com/hr/contact');
```

```
$sitemap->render()->send();
```

Save the changes. If you check your browser now, you will see:

¹⁶³<https://github.com/laravelista/creating-a-sitemap-with-bard-2-0/commit/77b94ee58ff68e9c3564ad4f9c621430187f46fb>

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9" xmlns:xhtml="http://\
www.w3.org/1999/xhtml">
  <url>
    <loc>http://domain.com</loc>
    <priority>1.0</priority>
    <changefreq>always</changefreq>
    <xhtml:link rel="alternate" hreflang="hr" href="http://domain.com/hr"/>
  </url>
  <url>
    <loc>http://domain.com/contact</loc>
    <priority>0.8</priority>
    <changefreq>hourly</changefreq>
    <lastmod>2017-07-05T10:48:57+00:00</lastmod>
    <xhtml:link rel="alternate" hreflang="hr" href="http://domain.com/hr/con\
tact"/>
  </url>
</urlset>
```

This is the current state of the repository at this moment [9c4acda49a9fa98354aa10f1d770e1dc4e790080](https://github.com/laravelista/creating-a-sitemap-with-bard-2-0/commit/9c4acda49a9fa98354aa10f1d770e1dc4e790080)¹⁶⁴.

Simplest example ever

The simplest example of creating a sitemap would be this:

```
<?php

require('vendor/autoload.php');

use Laravelista\Bard\UrlSet;
use Sabre\Xml\Writer;

$itemap = new UrlSet(new Writer);

$itemap->addUrl('http://domain.com');
$itemap->addUrl('http://domain.com/contact');

$itemap->render()->send();
```

and it would return:

¹⁶⁴<https://github.com/laravelista/creating-a-sitemap-with-bard-2-0/commit/9c4acda49a9fa98354aa10f1d770e1dc4e790080>

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9" xmlns:xhtml="http://\
www.w3.org/1999/xhtml">
  <url>
    <loc>http://domain.com</loc>
  </url>
  <url>
    <loc>http://domain.com/contact</loc>
  </url>
</urlset>
```

Thank you for reading. Stay tuned for the next tutorial on creating a sitemap with Bard and Laravel.

P.S. Give [Bard](#)¹⁶⁵ a star on GitHub.

¹⁶⁵<https://github.com/laravelista/Bard>

Bard 2.0 and Laravel

Using Bard with Laravel is much simpler than before. Let me show you.

Published at: 07. July, 2017.

A few days ago, I published a website [Visit Murter](https://visitmurter.com)¹⁶⁶ where you can explore and see the destinations on the island Murter. It is a relatively big website and I needed to create a sitemap for it. I already had a package for sitemap creation called [Bard](https://github.com/laravelista/Bard)¹⁶⁷, but at that moment the last commit to Bard was made in March 2015.

The package works and I have used it on many sites that I have created before. Even this site (Laravelista) uses Bard for Sitemap creation, but after working with it many times I have come to a conclusion that I can make it easier to get started with Bard.

I must admit, that every time that I did `composer require laravelista/bard` my next step was going to the documentation and figuring out how to implement the “Laravel boilerplate” that came with Bard. The boilerplate enabled you to easily implement translations and use route names instead of URLs. It was something that I was proud when I first created it, but soon I realized that it was a big pain in the ass.

This new version of Bard almost completely breaks the old way of doing things. Before you could have used the constructor to create the URL or you could have set only the location and then use multiple method calls to set the desired properties on the URL. In this version, there is only one way of doing things and that is by chaining methods. Also, setting translations has also been simplified.

In this tutorial, I will show you how to create an example sitemap using Bard and Laravel.

New project

From the place where you keep your repositories aka projects run this command to create a new Laravel application:

```
laravel new bard-laravel
```

If you don't have the `laravel` command installed on your PC, see the documentation on [Installing Laravel Via Laravel Installer](https://laravel.com/docs/5.4#installing-laravel)¹⁶⁸.

This will create a fresh Laravel installation with all of Laravel's dependencies already installed in the directory `bard-laravel`. When the installation completes you will see this message at the end:

¹⁶⁶<https://visitmurter.com>

¹⁶⁷<https://github.com/laravelista/Bard>

¹⁶⁸<https://laravel.com/docs/5.4#installing-laravel>

Application ready! Build something amazing.

This is the current state of the repository at this moment [b0c084f5557abe0051e45b83ac6306eb8363677a](https://github.com/laravelista/bard-20-and-laravel/commit/b0c084f5557abe0051e45b83ac6306eb8363677a)¹⁶⁹.

Installation

Now to install Bard we need to run:

```
composer require laravelista/bard
```

Output:

```
$ composer require laravelista/bard
Using version ^2.0 for laravelista/bard
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing sabre/uri (1.2.1)
  Loading from cache

- Installing sabre/xml (0.4.3)
  Loading from cache

- Installing laravelista/bard (2.0.0)
  Loading from cache
```

This means that Bard is installed. Awesome!

This is the current state of the repository at this moment [c8deb333b0ec1542ab8679ea72cec915e35eda72](https://github.com/laravelista/bard-20-and-laravel/commit/c8deb333b0ec1542ab8679ea72cec915e35eda72)¹⁷⁰.

Creating a sitemap

In `routes/web.php` add this code:

¹⁶⁹ <https://github.com/laravelista/bard-20-and-laravel/commit/b0c084f5557abe0051e45b83ac6306eb8363677a>

¹⁷⁰ <https://github.com/laravelista/bard-20-and-laravel/commit/c8deb333b0ec1542ab8679ea72cec915e35eda72>

```
Route::get('sitemap.xml', function(\Laravelista\Bard\UrlSet $sitemap) {
    return 'Hello';
});
```

Now run `php artisan serve` to test that everything is working. Point your browser to `http://localhost:8000/sitemap.xml`. You should see Hello text displayed.

Great!

Now let's replace the line `return 'Hello';` with the following:

```
$sitemap->addUrl('http://domain.com')
    ->setPriority(1.0)
    ->setChangeFrequency('always')
    ->addTranslation('hr', 'http://domain.com/hr');

$sitemap->addUrl('http://domain.com/contact')
    ->setPriority(0.8)
    ->setChangeFrequency('hourly')
    ->setLastModification(Carbon::now())
    ->addTranslation('hr', 'http://domain.com/hr/contact');

return $sitemap->render();
```

Save the changes. If you check your browser now, you will see:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9" xmlns:xhtml="http://\
www.w3.org/1999/xhtml">
  <url>
    <loc>http://domain.com</loc>
    <priority>1.0</priority>
    <changefreq>always</changefreq>
    <xhtml:link rel="alternate" hreflang="hr" href="http://domain.com/hr"/>
  </url>
  <url>
    <loc>http://domain.com/contact</loc>
    <priority>0.8</priority>
    <changefreq>hourly</changefreq>
    <lastmod>2017-07-07T10:53:03+00:00</lastmod>
    <xhtml:link rel="alternate" hreflang="hr" href="http://domain.com/hr/con\
tact"/>
  </url>
</urlset>
```

This is the current state of the repository at this moment [30738c9bc56e9eed702750258b0982bb0615f920](https://github.com/laravelista/bard-20-and-laravel/commit/30738c9bc56e9eed702750258b0982bb0615f920)¹⁷¹.

Simplest example ever

The simplest example of creating a sitemap would be this:

```
Route::get('sitemap.xml', function(\Laravelista\Bard\UrlSet $sitemap) {
    $sitemap->addUrl('http://domain.com');
    $sitemap->addUrl('http://domain.com/contact');

    return $sitemap->render();
});
```

and it would return:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9" xmlns:xhtml="http://\
www.w3.org/1999/xhtml">
    <url>
        <loc>http://domain.com</loc>
    </url>
    <url>
        <loc>http://domain.com/contact</loc>
    </url>
</urlset>
```

Thank you for reading.

P.S. Give [Bard](https://github.com/laravelista/Bard)¹⁷² a star on GitHub.

¹⁷¹<https://github.com/laravelista/bard-20-and-laravel/commit/30738c9bc56e9eed702750258b0982bb0615f920>

¹⁷²<https://github.com/laravelista/Bard>

Multilingual Web Application with Laravel



View Source Code

Source code for this course is available [here](#)¹⁷³.

Static Content

Let's start with the basics. Learn how to change your application locale, write localization files and use translations in your Laravel application.

Published at: 16. April, 2017.

Greetings! *You are about to embark on a great journey.*

If English is not your mother language, chances are that at some point in your career you will be tasked with creating a multilingual application/website. Lately, that is all that I have been doing. Creating a multilingual application complicates things and your code gets complex.

This course will guide you through the whole process of creating a multilingual application.

We will cover:

- static content / localization
- language switcher / routing (SEO)
- model translations / database

In the first part of this course aka this tutorial aka *Static Content*, you will learn how to change your application locale dynamically and statically, write localization file and use translations in your views.

Btw. You will also learn some Croatian ;)

This if the starting point for this tutorial [8ae5aedfde3ce53af160e3127e761523be90d2ae](#)¹⁷⁴

¹⁷³<https://github.com/laravelista/multilingual-web-application-with-laravel>

¹⁷⁴<https://github.com/laravelista/multilingual-web-application-with-laravel/commit/8ae5aedfde3ce53af160e3127e761523be90d2ae>

Changing Locale

There are two ways of changing your application locale: statically (changing the locale in the config file) and dynamically (setting the locale during code execution).

In `config/app.php` under *Application Locale Configuration* you will find your current application locale `en`. This value gets loaded into your application at application start.

The application locale determines the default locale that will be used by the translation service provider. You are free to set this value to any of the locales which will be supported by the application.

Bellow it you will see *Application Fallback Locale*. This value tells your application to look for a translation in this locale if a translation in current locale is not found.

The fallback locale determines the locale to use when the current one is not available. You may change the value to correspond to any of the language folders that are provided through your application.

To statically change your application locale change the value under *Application Locale Configuration* to anything you want. For now, we will leave it be (`en`).

To change your application locale during code execution (dynamically) type `\App::setLocale('hr');`. You can replace `hr` with the locale that you want.

Writing localization files

Your application localization files are located under `resources/lang/{locale}`. By default, Laravel comes with `en` locale directory with localization files for auth, pagination, passwords and validation.

If you start our application with `php artisan serve` and visit `/` in your browser, you will see:



Welcome page

We want to localize words Documentation and News.

Let's create a new file inside resources/lang/en called app.php. Paste this code inside:

```
<?php

return [
    // key => value, goes here
    'documentation' => 'Documentation',
    'news' => 'News'
];
```

We now have English translations for documentation and news. Let's create Croatian translation for the same words. In resources/lang create a new folder called hr. Inside it create a new file app.php and paste this code inside:

```
<?php

return [
    // key => value, goes here
    'documentation' => 'Dokumentacija',
    'news' => 'Novosti'
];
```

As you may have noticed, everything is the same as in the English localization file, except the right side (value). This is how you create localization files for different languages.

Using translations in views

There are two main ways of using a translation in your view:

- Blade directive @lang('app.documentation')
- function {{ trans('app.documentation') }}

To localize the words in our view, go to resources/views/welcome.blade.php and replace these lines:

```
<div class="links">
    <a href="https://laravel.com/docs">Documentation</a>
    <a href="https://laracasts.com">Laracasts</a>
    <a href="https://laravel-news.com">News</a>
    <a href="https://forge.laravel.com">Forge</a>
    <a href="https://github.com/laravel/laravel">GitHub</a>
</div>
```

with:

```
<div class="links">
    <a href="https://laravel.com/docs">@lang('app.documentation')</a>
    <a href="https://laravel-news.com">@lang('app.news')</a>
</div>
```

If you visit our application on /, you will see that it still says Documentation and News. That is because our application locale is set to en. If you change it to hr, the words will change. Let's do that now. Go to config/app.php and change locale to hr. View the site:



Welcome translated

Great! This completes this tutorial. You now have the basic knowledge of localization that comes with Laravel out-of-the-box.

You can also dynamically change the locale in `routes/web.php`. Before the return view statement, type `\App::setLocale('hr');`.

This is the current state of the repository at this moment [ef81e57d780211d52e324cf3218041e28e4405f6](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/ef81e57d780211d52e324cf3218041e28e4405f6)¹⁷⁵.

In the next tutorial, we will create a language switcher so that our users can choose the language that they want. Also, there will be talk about search engine optimization and routes. Happy Easter!

Language Switcher

We will implement a way for users to choose a language that they want.

Published at: 20. April, 2017.

In the previous tutorial, you have learned how to change your application locale and provide translated content. In this tutorial you will learn how to enable users to change the language of the website.

There are actually two ways of doing this:

¹⁷⁵<https://github.com/laravelista/multilingual-web-application-with-laravel/commit/ef81e57d780211d52e324cf3218041e28e4405f6>

- building the feature yourself
- using a package

Creating a language switcher is very simple and you could probably write it yourself. The whole process would go like this. Have a config file with languages supported, look for the locale in the URL, call `\App::setLocale('hr');` depending on the locale found in the URL. There ain't much to it.

With some failures along the way, you could probably pull it off. I have tried doing so in Laravel 4.2 and have burned myself in the end. Since then I have switched to using a package [mcamara/laravel-localization](https://github.com/mcamara/laravel-localization)¹⁷⁶.

In this tutorial, we will use the above-mentioned package to provide a language switcher, have routes that can be translated and routes that do not need to be translated.

Installation

Let's start by installing the package:

```
composer require mcamara/laravel-localization
```

Now, in `config/app.php` append to providers key:

```
Mcamara\LaravelLocalization\LaravelLocalizationServiceProvider::class
```

Add append to aliases:

```
'LaravelLocalization' => Mcamara\LaravelLocalization\Facades\LaravelLocalization\
::class
```

Finally, we have to register route middleware in `app/Http/Kernel.php`. Append to `$routeMiddleware` array:

¹⁷⁶<https://github.com/mcamara/laravel-localization>

```
'localize' => \Mcamara\LaravelLocalization\Middleware\LaravelLocalizationRoutes:\
:class,
'localizationRedirect' => \Mcamara\LaravelLocalization\Middleware\LaravelLocaliz\
ationRedirectFilter::class,
'localeSessionRedirect' => \Mcamara\LaravelLocalization\Middleware\LocaleSession\
Redirect::class
```

Configuration

Before we can declare our routes, we have to tweak a few configuration options for the package. We need to declare which locales our application supports.

There are two ways of configuring this package.

- Publishing the package config file with `php artisan vendor:publish --provider="Mcamara\LaravelLocal`. You can find the config file in `config/laravellocalization.php`.
- Overriding default config file values in a service provider

Service Provider

You can override config values in `app/Providers/AppServiceProvider` or in a new service provider `ConfigServiceProvider`. You can create it with `php artisan make:provider ConfigService-Provider`. Remember to register it in `config/app.php` under providers if you decide to go that way.

In this tutorial, we will override default config values for this package in `app/Providers/AppService-Provider`.

To view options that can be changed see the default config file for this package [here](#)¹⁷⁷.

Add this code under register method in `app/Providers/AppServiceProvider`:

```
config([
    'laravellocalization.supportedLocales' => [
        'en' => ['name' => 'English', 'script' => 'Latn', 'native' => 'English', \
'regional' => 'en_GB'],
        'hr' => ['name' => 'Croatian', 'script' => 'Latn', 'native' => 'Hrvatski\
', 'regional' => 'hr_HR'],
    ],
    'laravellocalization.hideDefaultLocaleInURL' => true,
]);
```

¹⁷⁷<https://raw.githubusercontent.com/mcamara/laravel-localization/master/src/config/config.php>

Here we are declaring that our application supports two locales `en` and `hr` and that it hides the default locale in the URL. The default locale for our application is set to `hr` in `config/app.php`. So if we are viewing our application in Croatian, the URL will not have the locale inside it.

Example:

```
http://localhost:8000/
http://localhost:8000/en

http://localhost:8000/fruits
http://localhost:8000/en/fruits
```

Routes

We now have everything ready to set up our routes.

SEO Advice

According to Google, the best way to organize your multilingual URLs is to prepend the URI with the locale. If you omit the locale from the URL, search engines will not know that you offer your content in multiple languages. You can also put the locale as the subdomain, but that makes everything more complex.

One more thing. On each page that can be viewed on another locale you have to provide a link to the URL for each locale that it supports. *This is solved with a language switcher.*

In `routes/web.php` wrap our one existing route:

```
Route::get('/', function () {
    return view('welcome');
});
```

with:

```
Route::group([
    'prefix' => LaravelLocalization::setLocale(),
    'middleware' => [ 'localizationRedirect' ]
], function () {
    /** ADD ALL LOCALIZED ROUTES INSIDE THIS GROUP **/
    Route::get('/', function () {
        return view('welcome');
    });
});

/** OTHER PAGES THAT SHOULD NOT BE LOCALIZED **/
```

Now if you visit `http://localhost:8000` you will see content on Croatian. If you visit `http://localhost:8000/en` you will see the same content on the English language.

Place every route that you want to support localization inside this group.

Nontranslatable routes

Sometimes, there are routes that you do not need translations for. An example for this would be the POST route for `/contact`. Place this routes outside the group we declared in our `routes/web.php` file.

Language switcher

We now have a route that can be viewed on both locales. To complete this tutorial we will now create a language switcher that will enable the users to choose a language in which they want to see the content in.

For this tutorial, we will keep it simple and place the switcher directly in `resources/views/welcome.blade.php` file. In the next tutorial, we will build something more complex.

Open `resources/views/welcome.blade.php` and replace this:

```
@if (Route::has('login'))
    <div class="top-right links">
        @if (Auth::check())
            <a href="{{ url('/home') }}">Home</a>
        @else
            <a href="{{ url('/login') }}">Login</a>
            <a href="{{ url('/register') }}">Register</a>
        @endif
    </div>
@endif
```

with

```
<div class="top-right links">
    @foreach(LaravelLocalization::getSupportedLocales() as $localeCode => $properties)
        <a rel="alternate" hreflang="{{ $localeCode }}" href="{{LaravelLocalization::getLocalizedURL($localeCode) }}"
            {{ $properties['native'] }}
        </a>
    @endforeach
</div>
```

Notice attributes `rel` and `hreflang`. These are important for SEO.

This piece of code adds a link for each locale that this application supports (remember `app/Providers/AppServiceProvider` register method).

You can now visit the website in the browser now and click on each locale to see how the translation and the URL changes.

ENGLISH HRVATSKI



Welcome page with a language switcher

This is the current state of the repository at this moment [5bee7d06fa7eb004348ad198d668bf301fcb578a](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/5bee7d06fa7eb004348ad198d668bf301fcb578a)¹⁷⁸.

Combining this tutorial with the previous one enables you to create multilingual websites that are SEO friendly. In the next tutorial, we will build upon this tutorial to build something more complex. **Model translations.**

Model Translations

Handling multilingual models has never been easier.

Published at: **01. May, 2017.**

There are a few ways of creating a multilingual database, which can be summarized to only two scenarios: build it yourself or use an existing package. Guess which one we will be doing in this tutorial :)

¹⁷⁸<https://github.com/laravelista/multilingual-web-application-with-laravel/commit/5bee7d06fa7eb004348ad198d668bf301fcb578a>

We will use this package [dimsav/laravel-translatable](https://github.com/dimsav/laravel-translatable)¹⁷⁹ for multilingual models. The readme in the package actually tells you how you need to write the migrations for your models for it to work correctly.

In the past, I have created my own way for multilingual models/database which worked, but after seeing this package and how it works, it just does not make any sense not to use it. It saves you so much time, as you will see in this tutorial, so let's get started.

We will create a model called `Article` which will have fields `title` and `content`. These fields need to be translatable to Croatian and English. This process will consist of us installing the package, writing migrations and setting up the model(s).

```
articles
- id INT UNSIGNED
- title VARCHAR
- content TEXT NULLABLE
- created_at DATETIME
- updated_at DATETIME
```

Package Installation

Install the package by running this command from the command line:

```
composer require dimsav/laravel-translatable
```

Output:

```
$ composer require dimsav/laravel-translatable
Using version ^7.0 for dimsav/laravel-translatable
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
  - Installing dimsav/laravel-translatable (v7.0)
    Loading from cache

Writing lock file
Generating optimized autoload files
> Illuminate\Foundation\ComposerScripts::postUpdate
> php artisan optimize
Generating optimized class loader
The compiled services file has been removed.
```

Now, add the service provider in `config/app.php`:

¹⁷⁹ <https://github.com/dimsav/laravel-translatable>

```
Dimsav\Translatable\TranslatableServiceProvider::class
```

Installation complete! Let's move on.

Writing Migrations

For each table that we want to have translations for we will need an extra table. In our case we need translations for articles table, so we will need a table called `article_translations`.

Let's create a new model with Migration by using the command line:

```
php artisan make:model Article -m
```

Output:

```
$ php artisan make:model Article -m
Model created successfully.
Created Migration: 2017_04_30_231628_create_articles_table
```

This command generates a model with the migration for us to populate for that model.

Open `database/migrations/2017_04_30_231628_create_articles_table.php` file and in the `up` method you will see:

```
Schema::create('articles', function (Blueprint $table) {
    $table->increments('id');
    $table->timestamps();
});
```

The above command also generates this boilerplate for us. This is everything that our articles table needs to have. We will now create a new table for `article_translations`. In the same `up` method, below the code for creating articles, paste the following:

```
Schema::create('article_translations', function(Blueprint $table)
{
    $table->increments('id');
    $table->integer('article_id')->unsigned();
    $table->string('title');
    $table->string('content')->nullable();
    $table->string('locale')->index();

    $table->unique(['article_id', 'locale']);
    $table->foreign('article_id')->references('id')->on('articles')->onDelete('c\
ascade');
});
```

The above piece of code will create a new table called `article_translations` which will hold the translations for our articles (title and content) for each locale. We have set the content column to be nullable because sometimes you don't want to enter the content immediately.

There can only be one article and locale combination in that database table (unique constraint). Also, if you delete the article, all related translations for that article are automatically deleted.

Important!

Modify the down method so that it contains:

```
Schema::dropIfExists('article_translations');
Schema::dropIfExists('articles');
```

Before we run the migrations, we will need to setup a database :)

Database setup

1. Create a new file called `database.sqlite` under database directory.
2. In your `.env` file set the value of `DB_CONNECTION` to `sqlite` (`DB_CONNECTION=sqlite`)

Run migrations with:

```
php artisan migrate
```

Output:


```
$ php artisan migrate
Migration table created successfully.
Migrating: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_000000_create_users_table
Migrating: 2014_10_12_100000_create_password_resets_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrating: 2017_04_30_231628_create_articles_table
Migrated: 2017_04_30_231628_create_articles_table
```

Awesome! We now have a database for multilingual articles.

Setting up models

If you remember, the command that we used to create the migration `php artisan make:model Article -m` also created a model `Article` for us under `app/`.

Open `app/Article.php` and add a trait:

```
use \Dimsav\Translatable\Translatable;
```

Then, we have to declare which fields are being translated in the “translation” model. We do that by adding a property called `translatedAttributes` and specifying the field names in an array:

```
public $translatedAttributes = ['title', 'content'];
```

Now, our `app/Article.php` file looks like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Article extends Model
{
    use \Dimsav\Translatable\Translatable;

    public $translatedAttributes = ['title', 'content'];
}
```

Translation Model

We need to create a new model for our Article translations called `ArticleTranslations`. We do that from the command line:

```
php artisan make:model ArticleTranslations
```

You can find the model in app/. Open it and add this code inside:

```
public $timestamps = false;
protected $fillable = ['title', 'content'];
```

Now our app/ArticleTranslations.php file looks like this:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class ArticleTranslations extends Model
{
    public $timestamps = false;
    protected $fillable = ['title', 'content'];
}
```

One last thing to configure before.

Configuration

To avoid publishing the config file, we will use the same approach that we used for setting the config values for the localization package.

Open file app/Providers/AppServiceProvider and in register method you will see:

```
config([
    'laravellocalization.supportedLocales' => [
        'en' => ['name' => 'English', 'script' => 'Latn', 'native' => 'English', \
        'regional' => 'en_GB'],
        'hr' => ['name' => 'Croatian', 'script' => 'Latn', 'native' => 'Hrvatski \
        ', 'regional' => 'hr_HR'],
    ],
    'laravellocalization.hideDefaultLocaleInURL' => true,
]);
```

If you didn't follow the previous tutorial, you will not see this code. Just continue reading.

Append to the end of that array:

```
'translatable.locales' => [
    'en',
    'hr'
],
```

That is it! Our `Article` model is now translatable to `en` and `hr` locale; both `title` and `content` fields.

This is the current state of the repository at this moment [aec5267191c50ffe233b6b2b21a8552dc7d9ba85](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/aec5267191c50ffe233b6b2b21a8552dc7d9ba85)¹⁸⁰.

In the next tutorial we will cover:

- saving translated attributes
- getting translated attributes
- filling multiple translations

Multilingual CRUD

In this tutorial, we will cover creating, updating, deleting and viewing all multilingual articles.

Published at: **06. May, 2017.**

Welcome to the latest part of this course. This tutorial will cover:

- saving translated attributes
- getting translated attributes
- filling multiple translations

We will create an `ArticleController` controller, create a resource route for it, and implement: `index`, `create`, `store`, `edit`, `update` and `destroy` methods.

When you finish reading this tutorial/course you will have all the necessary knowledge to build your own multilingual application with Laravel.

Controller

This is the current state of the repository at this moment [aec5267191c50ffe233b6b2b21a8552dc7d9ba85](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/aec5267191c50ffe233b6b2b21a8552dc7d9ba85)¹⁸¹.

Let's start by creating our `ArticleController`. From the command line type:

¹⁸⁰ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/aec5267191c50ffe233b6b2b21a8552dc7d9ba85>

¹⁸¹ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/aec5267191c50ffe233b6b2b21a8552dc7d9ba85>

```
php artisan make:controller ArticleController --resource
```

Great! Now, open the controller `app/Http/Controllers/ArticleController.php`, locate the `show` method and delete it. We won't be using it for this tutorial.

Route

Now it's time to create a resource route for our controller. We do that by going to the `routes/web.php` file and pasting this code at the end of the file (**Bellow the `/** OTHER PAGES THAT SHOULD NOT BE LOCALIZED */` comment**):

```
Route::resource('articles', 'ArticleController', ['except' => ['show']]);
```

We are placing the article resource route outside of our localized routes group because for this tutorial we will keep the “admin” part of our application on one language.

If you do `php artisan route:list` at this point, you should see:

```
$ php artisan route:list
```

Domain	Method	URI	Name	Action
	GET HEAD	/		Closure
	GET HEAD	api/user		Closure
	GET HEAD	articles	articles.index	App\Http\Controllers\ArticleController@index
	POST	articles	articles.store	App\Http\Controllers\ArticleController@store
	GET HEAD	articles/create	articles.create	App\Http\Controllers\ArticleController@create
	PUT PATCH	articles/{article}	articles.update	App\Http\Controllers\ArticleController@update
	DELETE	articles/{article}	articles.destroy	App\Http\Controllers\ArticleController@destroy
	GET HEAD	articles/{article}/edit	articles.edit	App\Http\Controllers\ArticleController@edit

Routes starting with `articles` are the ones that we are interested in this tutorial.

This is the current state of the repository at this moment [9fdcd6418919becf8094c01cba1d13e7a69b69c1](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/9fdcd6418919becf8094c01cba1d13e7a69b69c1)¹⁸².

Methods

For our `ArticleController` we will be using [Route Model Binding](https://laravel.com/docs/5.4/routing#route-model-binding)¹⁸³. To do that we need to replace any mention of `$id` in our methods parameters with `Article $article`. To this for `edit`, `update` and `destroy` methods. Also, remember to add `use App\Article;` statement above the class declaration of `ArticleController`.

This is the current state of the repository at this moment [9f6b9f28cab40b563e31b79e7ac3f6def8058636](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/9f6b9f28cab40b563e31b79e7ac3f6def8058636)¹⁸⁴.

Index

Let's begin!

In `app/Http/Controllers/ArticleController` under `index` method place these lines of code:

```
$articles = Article::latest()->get();

return view('articles.index')->with(compact('articles'));
```

The first line fetches the latest articles from the database. The second line returns a view file located in `resources/views/articles/index.blade.php` along with the fetched articles.

The view file mentioned in the code above does not exist yet. We have to create it. Under `resources/views/` create a new folder called `articles` and inside it create a file called `index.blade.php`.

Bootstrap layout view

To speed up the development we will be using [Bootstrap](http://getbootstrap.com)¹⁸⁵. To avoid copying the same boilerplate html template in every view, we will create a layout view in which we will place Bootstrap boilerplate with a few tweaks.

In `resources/views/` create a new folder called `layouts` and inside it create a file called `bootstrap.blade.php`. Inside that file place the following code:

¹⁸² <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/9fdcd6418919becf8094c01cba1d13e7a69b69c1>

¹⁸³ <https://laravel.com/docs/5.4/routing#route-model-binding>

¹⁸⁴ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/9f6b9f28cab40b563e31b79e7ac3f6def8058636>

¹⁸⁵ <http://getbootstrap.com>

```

<!DOCTYPE html>
<html lang="{{ App::getLocale() }}">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- The above 3 meta tags *must* come first in the head; any other head con\
tent must come *after* these tags -->
    <title>@yield('title')</title>

    <!-- Bootstrap -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7\
/css/bootstrap.min.css" integrity="sha384-BVYiisSIFeK1dGmJRAKycuHAHRg32OmUcww7on3\
RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">

    <!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media q\
ueries -->
    <!-- WARNING: Respond.js doesn't work if you view the page via file:// -->
    <!--[if lt IE 9]>
      <script src="https://oss.maxcdn.com/html5shiv/3.7.3/html5shiv.min.js"></sc\
ript>
      <script src="https://oss.maxcdn.com/respond/1.4.2/respond.min.js"></script>
    <![endif]-->
  </head>
  <body>
    @yield('content')

    <!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.\
js"></script>
    <!-- Include all compiled plugins (below), or include individual files as ne\
eded -->
    <script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.mi\
n.js" integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA7l2mCWNIpG9mGCD8wGN\
IcPD7Txa" crossorigin="anonymous"></script>
  </body>
</html>

```

Three things to notice here:

```

<html lang="{{ App::getLocale() }}">

```

This sets the language of the HTML document to the current locale of our application. Since our resource route for articles is placed outside of the translated routes group, the application locale will always be the default locale set in the configuration file: `hr` (View `config/app.php` to verify).

```
<title>@yield('title')</title>
```

This allows us to define the title of the page in each view that extends this layout.

```
@yield('content')
```

This is where the code for each view that extends this layout will be glued to.

```
*      *      *
```

Back to our `resources/views/articles/index.blade.php` file. Inside it paste the following content:

```
@extends('layouts.bootstrap')

@section('title', 'Articles Index')

@section('content')

@stop
```

Great! Now inside the content block paste the following code:

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      <p class="lead">Hello</p>
    </div>
  </div>
</div>
```

This above is the standard bootstrap syntax. We create a container, inside it a row and inside a row a column. Each row supports 12 columns. Sum of columns (space) in a row must match 12 or less. If you have no idea what I am talking about, give [Bootstrap: Getting Started](http://getbootstrap.com/getting-started/)¹⁸⁶ a read.

¹⁸⁶<http://getbootstrap.com/getting-started/>

What this above code does is simply output a big-ish Hello on the page.

What do we need on the index page?

We need to see all articles, a button to create a new article, a button to edit an existing article and a button to delete an article.

To create a button for creating a new article paste this code just below the container class, above the only row class:

```
<div class="row">
    <div class="col-md-12">
        <a class="btn btn-primary" href="{{ route('articles.create') }}">Create \
New</a>
    </div>
</div>
```

This creates a new row with a single column. Inside it is a link styled as a button that points to /articles/create URL.

To display all articles we need to loop through all of them (\$articles variable is available in our view) and display them in a way that makes the most sense. Replace <p class="lead">Hello</p> with:

```
@foreach($articles as $article)
    <div class="panel panel-default">
        <div class="panel-body">
            <h2>{{ $article->translate('en')->title }}</h2>
            <p>{{ str_limit($article->translate('en')->content, 200) }}</p>
        </div>
        <div class="panel-footer">
            <a class="btn btn-default btn-xs" href="{{ route('articles.edit', $a\
rticle->id) }}">Edit</a>
            <a class="btn btn-danger btn-xs" href="{{ route('articles.destroy', \
$article->id) }}" onclick="event.preventDefault();document.getElementById('artic\
le_{{ $article->id }}').submit();">Delete</a>
            <form id="article_{{ $article->id }}" action="{{ route('articles.des\
troy', $article->id) }}" method="POST" style="display: none;">
                <input type="hidden" name="_method" value="DELETE">
                {{ csrf_field() }}
            </form>
        </div>
    </div>
@endforeach
```



```
<p class="lead">No articles yet...</p>
@endforeach
```

Let's go step by step over this code above.

We use a Bootstrap *Panel* component to display each article. In the *body* section of the panel we display the title of the article on English and the content of the article on English, but only the first 200 characters. In the *footer* of the panel, we display two buttons. An *Edit* button and a *Destroy* button. The Destroy button has an `onClick` event set on it which submits the **hidden form** that sends a *DELETE* request to delete that article.

We could have used a normal GET request on the *Delete* button (and change the route in the routes file), but this way it is much safer. *A potential attacker could use JavaScript to send GET requests to the delete route with different IDs that will result in the application deleting those articles. In this way a user is required to send a valid csrf_token for each deletion.*

The index page is now complete. If you save the changes and view it in the browser (URI `/articles`) you will see the *Create New* button and a paragraph saying: "No articles yet...".

In the next chapter, we will create a *Create* view and handle article creation.

This is the current state of the repository at this moment [6f391c777830cddb7da0592af1d9219dec17b5b8](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/6f391c777830cddb7da0592af1d9219dec17b5b8)¹⁸⁷.

Create/store

Back to the `app/Http/Controllers/ArticleController.php`. Add this line to the create method:

```
return view('articles.create');
```

Now we have to create that view. In the `resources/views/articles/` create a new file called `create.blade.php`. In this view, we will create a form for creating a multilingual article. We want to enable the user to enter both translations at once (both `hr` and `en`). The content can be nullable, but the title is required on both translations.

Sometimes you may want to force that the user enters a single translation and that others be optional, but for this tutorial, both translations are required. Depending on what are your business needs you should adapt your application.

Paste this code inside `articles/create.blade.php` file:

¹⁸⁷ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/6f391c777830cddb7da0592af1d9219dec17b5b8>

```

@extends('layouts.bootstrap')

@section('title', 'Articles Create')

@section('content')

<div class="container">
    <div class="row">
        <div class="col-md-12">
            {{-- Code goes here --}}
        </div>
    </div>
</div>

@stop

```

You can notice that we leverage our bootstrap layout in this view also. Now replace the Blade comment `{{-- Code goes here --}}` with our form:

```

<div class="container">
    <div class="row">
        <div class="col-md-12">
            <form method="POST" action="{{ route('articles.store') }}">
                {{ csrf_field() }}
                @foreach(config('translatable.locales') as $locale)
                    <div class="form-group">
                        <label for="translation[{{ $locale }}][title]">Title ({{ $lo\
ocale }})</label>
                        <input type="text" class="form-control" name="translatio\
n[{{ $locale }}][title]">
                    </div>
                    <div class="form-group">
                        <label for="translation[{{ $locale }}][content]">Content (\
{{ $locale }})</label>
                        <textarea class="form-control" name="translation[{{ $loca\
le }}][content]"></textarea>
                    </div>
                @endforeach
                <button type="submit" class="btn btn-default">Create</button>
            </form>
        </div>
    </div>
</div>

```

Two things to notice here:

1. We create fields for all supported locales by our application as defined in the app/Providers/AppServiceProvider.php
2. We are using form arrays to simplify form validation for multilingual fields

One thing that is missing in our create view is a way to display validation errors. Since we will be using the logic for display validation errors on many pages in the future it is wise to place it in a partial that we can include in every view that requires it.

Create a file called _validation.blade.php in resources/views. Inside that file paste the following:

```
@if (count($errors) > 0)
    <div class="alert alert-danger">
        <ul>
            @foreach ($errors->all() as $error)
                <li>{{ $error }}</li>
            @endforeach
        </ul>
    </div>
@endif
```

Now in resources/views/articles/create.blade.php just above the form tag place this line:

```
@include('_validation')
```

Awesome! Our creation view is now complete and it looks like this:

The screenshot shows a web form for creating an article. It has four input fields: 'Title (en)', 'Content (en)', 'Title (hr)', and 'Content (hr)'. Each field is a text input with a small icon in the bottom right corner. Below the fields is a 'Create' button. The form is styled with a light gray border and a white background.

Create view

To handle article validation and creation we need to go to app/Http/Controllers/ArticleController.php and inside the store method place the following code to handle validation:

```
$this->validate($request, [
    'translation.*.title' => 'required|string',
    'translation.*.content' => 'nullable|string',
]);
```

This will validate all sent translations (hr and en in our case) for title and content. Now if you add more supported translations to your application in the future, the creation form will automatically generate appropriate fields and they will be properly validated by the controller.

If you want to learn more about form array validation read my tutorial on it here:
[Validating Form Arrays](#).

Append the following code to the store method below the validation block to handle article creation:

```
$article = new Article;
foreach (config('translatable.locales') as $locale) {
    $article->fill([
        $locale => [
            'title' => $request->get('translation')[$locale]['title'],
            'content' => $request->get('translation')[$locale]['content']
        ]
    ]);
}
$article->save();
```

We create an empty Article model which is then filled with the title and content for each translation as defined in the app/Providers/AppServiceProvider.php and persist it in the database.

Place the following code at the end of the store method to redirect to user to the index page after successful article creation:

```
return redirect()->route('articles.index');
```

If you fill the form and submit it you will receive an error at this point saying that there is no App/ArticleTranslation. You need to rename app/Http/ArticleTranslations.php to app/Http/ArticleTranslation.php and change the class name in the same file from ArticleTranslations to ArticleTranslation.

Try filling the form and submitting. You should be redirected to the index page where you will see the article that you have just created.

Also, try submitting an empty form to see that the validation errors work.

This is the current state of the repository at this moment [339d1a05890c0fb3ec0b07eae5b5dc2a33e36333](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/339d1a05890c0fb3ec0b07eae5b5dc2a33e36333)¹⁸⁸.

¹⁸⁸<https://github.com/laravelista/multilingual-web-application-with-laravel/commit/339d1a05890c0fb3ec0b07eae5b5dc2a33e36333>

Edit/Update

The editing process is almost the same as the creation process. In our index page we already have a button on each article that navigates us to the route which handles article update. First, we need to create a view which will display a form populated with the values of the article that we want to edit, and then we need to handle updating the article in the controller.

Go to `app/Http/Controllers/ArticleController.php` and in the edit method paste this line:

```
return view('articles.edit')->with(compact('article'));
```

Now we need to create that view. Create a new file called `edit.blade.php` in `resources/views/articles/` directory and paste the following code inside it:

```
@extends('layouts.bootstrap')

@section('title', 'Articles Edit')

@section('content')

<div class="container">
    <div class="row">
        <div class="col-md-12">
            <br />
            @include('_validation')
            <form method="POST" action="{ route('articles.update', $article->id\
) }}">

                <input type="hidden" name="_method" value="PUT">
                {{ csrf_field() }}
                @foreach(config('translatable.locales') as $locale)
                    <div class="form-group">
                        <label for="translation[{{$locale}}][title]">Title ({{$l\
ocale}})</label>
                        <input type="text" class="form-control" name="translatio\
n[{{$locale}}][title]" value="{ $article->translate($locale)->title }}">
                    </div>
                    <div class="form-group">
                        <label for="translation[{{$locale}}][content]">Content (\
{{$locale}})</label>
                        <textarea rows="5" class="form-control" name="translatio\
n[{{$locale}}][content]">{{ $article->translate($locale)->content }}</textarea>
                    </div>
                @endforeach
            </div>
        </div>
    </div>
</div>
```

```

        <button type="submit" class="btn btn-default">Update</button>
    </form>
</div>
</div>
</div>

@stop

```

Great! Now in the update method of the app/Http/Controllers/ArticleController.php add this code to handle validation:

```

$this->validate($request, [
    'translation.*.title' => 'required|string',
    'translation.*.content' => 'nullable|string',
]);

```

It is the same as in the store method. Below that, add this code to update the article with new values:

```

foreach (config('translatable.locales') as $locale) {
    $article->fill([
        $locale => [
            'title' => $request->get('translation')[$locale]['title'],
            'content' => $request->get('translation')[$locale]['content']
        ]
    ]);
}
$article->save();

```

Here we omit the `$article = new Article;` line because `$article` is resolved from the URL by using *Route Model Binding* as mentioned at the start of this tutorial. Finally, we will redirect the user back to the index page with this line:

```
return redirect()->route('articles.index');
```

You can choose if you want to redirect the user to the index page, back to the edit page or any other page you want.

Test it all out to see that it works and let's move on to destroying articles :)

This is the current state of the repository at this moment [0faf0522f63f37a38b44f74146cee38edb77a545](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/0faf0522f63f37a38b44f74146cee38edb77a545)¹⁸⁹.

¹⁸⁹ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/0faf0522f63f37a38b44f74146cee38edb77a545>

Destroy

Destroying an article with all of its translations is super simple. In the destroy method, just write these two lines:

```
$article->delete();

return redirect()->route('articles.index');
```

This is the current state of the repository at this moment [f6ed94567ec283df1eb7b021bd66587435cb1703](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/f6ed94567ec283df1eb7b021bd66587435cb1703)¹⁹⁰.

* * *

Thank you for reading this tutorial.

You now know how to handle multilingual models!

In the next part, we will setup authentication so that only the authenticated users can use our “admin” part of this website to create, update and delete articles. This is entirely optional and has nothing to do with multilingual models, but to complete this application it is a necessity.

Authentication and Authorization

Laravel makes implementing authentication very simple. We will implement authentication in our existing application.

Published at: 22. May, 2017.

In the previous parts of this course, we have built a multilingual web application using Laravel and a couple of packages. We have set up a route `/articles` on which we can see all articles, create, edit and delete articles. In this tutorial, we will implement authentication so that only the authenticated users can access routes behind `/article` URI.

We will also have to make a few changes to our layout so that the users can quickly access the most important parts of our application.

For demonstration purposes, we will allow everyone to register, but usually what you would want to do is to prevent that behavior for your “admin” section, or you can allow users that have permissions to access that part of your application. It depends on your business needs.

Let’s begin!

Authentication Quickstart

In Laravel 5.4 almost everything is configured for you out of the box. To get things started just run this command from the command line:

¹⁹⁰<https://github.com/laravelista/multilingual-web-application-with-laravel/commit/f6ed94567ec283df1eb7b021bd66587435cb1703>

```
php artisan make:auth
```

This command will create your entire authentication system.

Changes:

routes/web.php

Routes:

```
Auth::routes();

Route::get('/home', 'HomeController@index');
```

app/Http/Controllers/

Files:

- HomeController.php

resources/views/

Files:

- home.blade.php
- auth/*
- layouts/app.blade.php

Auth::routes() generates this routes for your application:

```
// Authentication Routes...
$this->get('login', 'Auth\LoginController@showLoginForm')->name('login');
$this->post('login', 'Auth\LoginController@login');
$this->post('logout', 'Auth\LoginController@logout')->name('logout');

// Registration Routes...
$this->get('register', 'Auth\RegisterController@showRegistrationForm')->name('register');
$this->post('register', 'Auth\RegisterController@register');

// Password Reset Routes...
$this->get('password/reset', 'Auth\ForgotPasswordController@showLinkRequestForm')->name('password.request');
$this->post('password/email', 'Auth\ForgotPasswordController@sendResetLinkEmail');
```



```

)->name('password.email');
$this->get('password/reset/{token}', 'Auth\ResetPasswordController@showResetForm\
')->name('password.reset');
$this->post('password/reset', 'Auth\ResetPasswordController@reset');

```

Since we have already migrated our database in the previous tutorial we don't have to migrate it now. Laravel comes with two migration files out of the box. Those two migrations contain instructions for creating the users and password_resets tables. If you haven't migrated your database yet, you can do that now with `php artisan migrate`.

This is the current state of the repository at this moment [72af558240bf4645dfa196244f248c8e8b30e1ce](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/72af558240bf4645dfa196244f248c8e8b30e1ce)¹⁹¹.

Tweaking

We will now modify our application to increase integration with the newly implemented authentication system.

Removing the junk

We won't be needing the `HomeController`, the `/home` route and the `resources/views/home.blade.php` file. First, let's delete the file `app/Http/Controllers/HomeController.php`. Then, inside the file `routes/web.php` locate the line `Route::get('/home', 'HomeController@index');` and delete it. Finally, delete the file `resources/views/home.blade.php`. Awesome!

This is the current state of the repository at this moment [a902cb24a67ee0d7f1dd12411ed1a575290fe5ed](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/a902cb24a67ee0d7f1dd12411ed1a575290fe5ed)¹⁹².

Providing login and register links

From our “welcome” page we want to provide our users links to login and register pages. Since our application is multilingual we need to use the knowledge from the first part of this course called *Static Content*.

First, in `resources/views/welcome.blade.php` below the “Documentation” and “News” links place the following code:

```

<a href="{{ url('/login') }}">@lang('app.login')</a>
<a href="{{ url('/register') }}">@lang('app.register')</a>

```

Then, we have to create translations for those two references in `resources/lang/en/app.php` for English language:

¹⁹¹ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/72af558240bf4645dfa196244f248c8e8b30e1ce>

¹⁹² <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/a902cb24a67ee0d7f1dd12411ed1a575290fe5ed>

```
'register' => 'Register',
'login' => 'Login'
```

and `resources/lang/hr/app.php` for Croatian language:

```
'register' => 'Registracija',
'login' => 'Prijava'
```

As you may have noticed if you visited `/login` or `/register` routes, those pages are only in the English language. Those pages can easily be made multilingual with the knowledge from this course. That is your homework.

This is the current state of the repository at this moment [d78ad9655662ee0868ac6ac26cd819aac5d73f84](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/d78ad9655662ee0868ac6ac26cd819aac5d73f84)¹⁹³.

Articles link for authenticated users

Once the user has logged in we want to display a link to *articles* to them. That link must be only visible to authenticated users.

In `resources/views/welcome.blade.php` find:

```
<a href="{{ url('/login') }}">@lang('app.login')</a>
<a href="{{ url('/register') }}">@lang('app.register')</a>
```

and replace it with the following code:

```
@if(Auth::check())
    <a href="{{ route('articles.index') }}">@lang('app.articles')</a>
@else
    <a href="{{ url('/login') }}">@lang('app.login')</a>
    <a href="{{ url('/register') }}">@lang('app.register')</a>
@endif
```

Add the translation to correct files for `app.articles`. English: *Articles*; Croatian: *Äœlanci*. The files are located in `resources/lang/{locale}/`.

The above code checks if the user is logged in and if it is logged in then it displays the *Articles* link. If the user is not logged in, it displays the *login* and *register* links.

This is the current state of the repository at this moment [a9d913635a706dea5b03a4546693fb85d457ad36](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/a9d913635a706dea5b03a4546693fb85d457ad36)¹⁹⁴.

¹⁹³ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/d78ad9655662ee0868ac6ac26cd819aac5d73f84>

¹⁹⁴ <https://github.com/laravelista/multilingual-web-application-with-laravel/commit/a9d913635a706dea5b03a4546693fb85d457ad36>

Register/Login Logout Flow

We need to change where the user is redirected after he logs in or registers. By default, the user is redirected to the /home route, but since we have deleted that route we need to change that behavior.

We do that by changing the values of \$redirectTo in app/Http/Auth/LoginController.php and app/Http/Auth/RegisterController from /home to /.

Let's try to register now. You should get redirected to / page. Now click on *Articles* (Articles) and on the next page try to locate the *Log Out* link. You can't find it, can you?

We will now copy the *navbar* section from the resources/views/layouts/app.blade.php to resources/views/layouts/bootstrap.blade.php:

```
<nav class="navbar navbar-default navbar-static-top">
  <div class="container">
    <div class="navbar-header">

      <!-- Collapsed Hamburger -->
      <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#app-navbar-collapse">
        <span class="sr-only">Toggle Navigation</span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
      </button>

      <!-- Branding Image -->
      <a class="navbar-brand" href="{{ url('/') }}">
        {{ config('app.name', 'Laravel') }}
      </a>
    </div>

    <div class="collapse navbar-collapse" id="app-navbar-collapse">
      <!-- Left Side Of Navbar -->
      <ul class="nav navbar-nav">
        <li>
          <a href="{{ route('articles.index') }}">Articles</a>
        </li>
      </ul>

      <!-- Right Side Of Navbar -->
      <ul class="nav navbar-nav navbar-right">
        <li class="dropdown">
```

```

        <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-expanded="false">
            {{ Auth::user()->name }} <span class="caret"></span>
        </a>

        <ul class="dropdown-menu" role="menu">
            <li>
                <a href="{{ route('logout') }}"
                    onclick="event.preventDefault();
                        document.getElementById('logout-form').submit();">
                    Logout
                </a>

                <form id="logout-form" action="{{ route('logout') }}" \
method="POST" style="display: none;">
                    {{ csrf_field() }}
                </form>
            </li>
        </ul>
    </li>
</ul>
</div>
</div>
</nav>

```

Awesome, now if you refresh the page you will see the *Log out* link in the drop-down menu on the right. Click on the *Log out* link now and open in browser URI `/articles`. As you can see still everyone can manage articles... we are going to change that in the following chapter.

This is the current state of the repository at this moment [6121ab1bb296680387d7109b3cdc435bc56b873c](https://github.com/laravelista/multilingual-web-application-with-laravel/commit/6121ab1bb296680387d7109b3cdc435bc56b873c)¹⁹⁵.

Protecting routes/controller

In this chapter, we will use auth middleware on the `ArticleController` to only authorize the authenticated users to perform actions and access routes.

If you do `php artisan route:list` now, you will see:

¹⁹⁵<https://github.com/laravelista/multilingual-web-application-with-laravel/commit/6121ab1bb296680387d7109b3cdc435bc56b873c>

```
$ php artisan route:list
```

```
+-----+-----+-----+-----+-----+-----+
| Domain | Method | URI | Name | Action | Middleware |
+-----+-----+-----+-----+-----+-----+
| GET|HEAD | / | | Closure | web, localizationRed |
| GET|HEAD | api/user | | Closure | api, auth:api |
| GET|HEAD | articles | articles.index | App\Http\Controllers\ArticleController@index | web |
| POST | articles | articles.store | App\Http\Controllers\ArticleController@store | web |
| GET|HEAD | articles/create | articles.create | App\Http\Controllers\ArticleController@create | web |
| PUT|PATCH | articles/{article} | articles.update | App\Http\Controllers\ArticleController@update | web |
| DELETE | articles/{article} | articles.destroy | App\Http\Controllers\ArticleController@destroy | web |
| GET|HEAD | articles/{article}/edit | articles.edit | App\Http\Controllers\ArticleController@edit | web |
| GET|HEAD | login | login | App\Http\Controllers\Auth\LoginController@showLoginForm | web, guest |
| POST | login | | App\Http\Controllers\Auth\LoginController@login | web, guest |
| POST | logout | App\Http\Controllers\Auth\LoginController@logout | web |
```

```

|
| POST | password/email | password.email | App\Http\Controllers\Auth\ForgotPasswordController@sendResetLinkEmail | web,guest |
|
| GET|HEAD | password/reset | password.request | App\Http\Controllers\Auth\ForgotPasswordController@showLinkRequestForm | web,guest |
|
| POST | password/reset | | App\Http\Controllers\Auth\ResetPasswordController@reset | web,guest |
|
| GET|HEAD | password/reset/{token} | password.reset | App\Http\Controllers\Auth\ResetPasswordController@showResetForm | web,guest |
|
| GET|HEAD | register | register | App\Http\Controllers\Auth\RegisterController@showRegistrationForm | web,guest |
|
| POST | register | | App\Http\Controllers\Auth\RegisterController@register | web,guest |
|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+

```

Make notice of the *Middleware* column.

Now in `app/Http/Controllers/ArticleController.php` add a constructor method and inside it attach the auth middleware like so:

```
public function __construct()
{
    $this->middleware('auth');
}
```

The code above attached the `auth` middleware to all methods inside the `ArticleController` class. Now if you do `php artisan route:list` you should see the `auth` middleware on all routes that are related to the `ArticleController`:

```
$ php artisan route:list
```

```
+-----+-----+-----+-----+-----+-----+
| Domain | Method | URI | Name | Action | Middleware |
|-----+-----+-----+-----+-----+-----+
| GET|HEAD | / | | Closure | web, localizationRed |
| GET|HEAD | api/user | | Closure | api, auth:api |
| GET|HEAD | articles | articles.index | App\Http\Controllers\ArticleController@index | web, auth |
| POST | articles | articles.store | App\Http\Controllers\ArticleController@store | web, auth |
| GET|HEAD | articles/create | articles.create | App\Http\Controllers\ArticleController@create | web, auth |
| PUT|PATCH | articles/{article} | articles.update | App\Http\Controllers\ArticleController@update | web, auth |
| DELETE | articles/{article} | articles.destroy | App\Http\Controllers\ArticleController@destroy | web, auth |
| GET|HEAD | articles/{article}/edit | articles.edit | App\Http\Controllers\ArticleController@edit | web, auth |
| GET|HEAD | login | login | App\Http\Controllers\Auth\LoginController@showLoginForm | web, guest |
| POST | login | | App\Http\Controllers\Auth\LoginController@login | web, guest |
| POST | logout | App\Http\Controllers\Auth\LoginController@logout | web |
```

```

|
| POST | password/email | password.email | App\Http\Controllers\Auth\ForgotPasswordController@sendResetLinkEmail | web,guest |
|
| GET|HEAD | password/reset | password.request | App\Http\Controllers\Auth\ForgotPasswordController@showLinkRequestForm | web,guest |
|
| POST | password/reset | | App\Http\Controllers\Auth\ResetPasswordController@reset | web,guest |
|
| GET|HEAD | password/reset/{token} | password.reset | App\Http\Controllers\Auth\ResetPasswordController@showResetForm | web,guest |
|
| GET|HEAD | register | register | App\Http\Controllers\Auth\RegisterController@showRegistrationForm | web,guest |
|
| POST | register | | App\Http\Controllers\Auth\RegisterController@register | web,guest |
|
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
-----+

```

Great! You can now try to access `/articles` while signed out. You should be redirected to the `/login` route. If you log in now and try visiting the same route, you should be able to see the articles index page as expected.

This is the current state of the repository at this moment [69cfee7079a17da381dc7750d808fe12c86f9b4d](#)¹⁹⁶.

This tutorial concludes this entire course on multilingual Laravel applications. Thank you for reading and have a great week.

P.S. This tutorial was a bit late because I finally graduated :D and had to go to Zagreb to attend the promotion and deal with the paperwork.

¹⁹⁶<https://github.com/laravelista/multilingual-web-application-with-laravel/commit/69cfee7079a17da381dc7750d808fe12c86f9b4d>

API Development

Building and documenting an API with Laravel.

Laravel API 101

Building an API is very simple once you get the hang of it. I'll show you how you to build an API, consume it, test it and write good documentation for it.

BUILD

In this tutorial, we will build an API using Laravel for managing news articles.

Published at: 23. May, 2016.



View Source Code

Source code for this tutorial is available [here](#)¹⁹⁷.

This tutorial is a successor to **Using Fractal with Laravel to create an API** and **Build an API with Lumen and Fractal** which were the most viewed posts on my blog. I've come a long way since then and now I will show you how to create an API that can index, store, show, update and destroy a resource.

- We will be using [Fractal](#)¹⁹⁸ as a presentation and transformation layer for our data
- To view the API output I will show you how to use [Postman](#)¹⁹⁹
- To avoid reinventing the wheel we will use a package I wrote called [Syndra](#)²⁰⁰ which provides common JSON responses for an API

I am sure that you will like this tutorial and that you will benefit from it. I have really made an effort to minimize the amount of code and steps needed to build an API.

In future tutorials, I will build upon this tutorial and show you how to implement a Token-Based Authentication, use this API in another application, write tests for it and finally create beautiful documentation.

¹⁹⁷<https://github.com/laravelista/laravel-api-101-build>

¹⁹⁸<http://fractal.thephpleague.com/>

¹⁹⁹<http://www.getpostman.com/>

²⁰⁰<https://github.com/laravelista/Syndra>

Common Questions

Here I will answer questions that I think are most common for beginners when starting working with APIs. Send me a question in comments below that you would like to see answered here and I will do my best to include it.

What is an API?

API stands for Application Programming Interface. An API provides *endpoints* to *resources*, and returns *responses* in JSON or XML format.

That is the most simple explanation that I could think of.

When to use an API?

There are many reasons when and where to use an API. The best I can answer this question is to use it where you find it appropriate. *Experience leads to wisdom.*

This tutorial will help you understand what is an API so that you can decide for yourself when and where to use it.

Why do I need to build an API?

Again, there are many reasons to why you would need to build an API.

For example, in my case, I maintain a big “old” application written in Laravel 4.2 which I cannot simply update to the latest version of Laravel and it also has some database design flaws. I needed to access the records from that application so that I could display them in another application.

To avoid duplicating models and responsibility I created an API and structured the data there how I wanted. So now when and if I update that old application I will just make sure that the API returns the same output and all of my applications that connect to it will still work without any changes needed.

Starting Point

I have prepared a Laravel 5.2.32 application with models, migrations, factories and seeder classes on GitHub that you should use as a starting point for this tutorial.

Be sure to read the `readme.md` file that comes with it because you will need to:

- generate app key
- create a database file
- migrate database

- seed the database with fake articles, tags and categories



View changes in this commit

[ac8a8e98c2e95a1eed5bcb5eb9ffd7bcd42bec5e](https://github.com/laravelista/laravel-api-101-build/commit/ac8a8e98c2e95a1eed5bcb5eb9ffd7bcd42bec5e)²⁰¹.

Install required packages

We will start by installing packages that will make the process of building an API easier and faster.

Fractal

“Fractal provides a presentation and transformation layer for complex data output, the like found in RESTful APIs, and works really well with JSON.” - Taken from Fractal [official website](https://fractal.thephpleague.com/)²⁰².

To install Fractal, type in command line:

```
composer require league/fractal
```

Output:

```
Using version ^0.13.0 for league/fractal
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing league/fractal (0.13.0)
  Downloading: 100%
```

Syndra

“Common JSON responses for an API built with Laravel” - Taken from Syndra [official website](https://github.com/laravelista/Syndra)²⁰³.

Whenever I build an API I use this package because it saves me lots of time. You can easily modify it to your needs, but in 99% cases, it already has everything you need.

To install Syndra, type in command line:

²⁰¹<https://github.com/laravelista/laravel-api-101-build/commit/ac8a8e98c2e95a1eed5bcb5eb9ffd7bcd42bec5e>

²⁰²<http://fractal.thephpleague.com/>

²⁰³<https://github.com/laravelista/Syndra>

```
composer require laravelista/syndra
```

Output:

```
Using version ^1.2 for laravelista/syndra
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing laravelista/syndra (1.2.1)
  Loading from cache
```

Now we have to add the service provider to our application in `config/app.php`:

```
'providers' => [
    ...
    Laravelista\Syndra\SyndraServiceProvider::class
];
```

And finally let's add a facade alias for it:

```
'aliases' => [
    ...
    'Syndra' => Laravelista\Syndra\Facades\Syndra::class
];
```

Postman

[Postman](http://www.getpostman.com/)²⁰⁴ is not a PHP package that you can install in your application. It is a Google Chrome Standalone application that you install from the [Google Chrome Web Store](https://chrome.google.com/webstore/detail/postman-rest-client/fhbjgbiflinjbdggehcddcbncdddomop)²⁰⁵ or as a [Mac App](https://www.getpostman.com/app/postman-osx-beta?utm_source=site&utm_medium=homepage&utm_campaign=macapp)²⁰⁶.

You will need Postman in this tutorial to view the output from the API that we will build and to use it for authenticating to the API. To learn more about Postman and what it does [click here](http://www.getpostman.com/)²⁰⁷.

Setting up routes

Go to file `app/Http/routes.php` and add the following:

²⁰⁴<http://www.getpostman.com/>

²⁰⁵<https://chrome.google.com/webstore/detail/postman-rest-client/fhbjgbiflinjbdggehcddcbncdddomop>

²⁰⁶https://www.getpostman.com/app/postman-osx-beta?utm_source=site&utm_medium=homepage&utm_campaign=macapp

²⁰⁷<http://www.getpostman.com/>

```
Route::group(['prefix' => 'api/v1'], function ()
{
    Route::resource('articles', 'ArticleController', ['except' => [
        'create', 'edit'
    ]]);
});
```

Here we added a route group which we prefixed with `api/v1` and inside it, we created a resource route for articles that is handled by `ArticleController`. Because we are building an API, we will not need routes for displaying the create and edit forms. That is why we excluded them from the resource routes that we defined.

Before we continue, we will create a [RESTful Resource Controller](#)²⁰⁸ for articles called `ArticleController` using this command:

```
php artisan make:controller ArticleController --resource
```

The controller will contain a method for each of the available resource operations.

From the `ArticleController` delete methods: `create` and `edit` because we will not need them.

To see which routes are available now in our application we can use:

```
php artisan route:list
```

Output:

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Domain | Method | URI | Name | Middleware | Action |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|      | GET|HEAD | api/v1/articles | api.v1.articles.index | web | Ap\
p\Http\Controllers\ArticleController@index |
|      | POST | api/v1/articles | api.v1.articles.store | web | Ap\
p\Http\Controllers\ArticleController@store |
|      | GET|HEAD | api/v1/articles/{articles} | api.v1.articles.show | web | Ap\
p\Http\Controllers\ArticleController@show |
|      | PUT|PATCH | api/v1/articles/{articles} | api.v1.articles.update | web | Ap\
p\Http\Controllers\ArticleController@update |
```

²⁰⁸<https://laravel.com/docs/5.2/controllers#restful-resource-controllers>

```
|          | DELETE      | api/v1/articles/{articles} | api.v1.articles.destroy | Ap\
p\Http\Controllers\ArticleController@destroy | web          |
+-----+-----+-----+-----+-----+-----+-----+
-----+-----+
```

This is the current state of the repository at this moment [1e9c47432647bd77a1e2bd55ebcdf559e7344cac](https://github.com/laravelista/laravel-api-101-build/tree/1e9c47432647bd77a1e2bd55ebcdf559e7344cac)²⁰⁹.

We are now ready to start building our API.

Building an API

To sum things up. At this point we have installed the packages that we need to build an API, we have configured our routes, we have created a resource controller for articles. The preparations are all complete, we can move on to the fun part now.

In this part, I will show you how to use Fractals transformer classes to represent article data, use default and optional includes, use Syndra to return correct responses and finally I'll show you how to store, update and destroy articles.

Article Endpoints

Everything in our little imaginary application revolves around articles, so it is only logical to build that endpoint first. Later you can add endpoints for categories or tags if you want to.

Index

Before we start with the index method, we first have to create a constructor and add dependencies for the two packages we installed previously: Fractal and Syndra and finally add a dependency for articles model.

Open file `app/Http/Controllers/ArticleController` and create a constructor for it like so:

```
protected $fractal;
protected $syndra;
protected $articles;

public function __construct(Manager $fractal, Syndra $syndra, Article $articles)
{
    $this->fractal = $fractal;
    $this->syndra = $syndra;
    $this->articles = $articles;
}
```

Also be sure to **use** those classes at the top of the file, just below the namespace definition:

²⁰⁹ <https://github.com/laravelista/laravel-api-101-build/tree/1e9c47432647bd77a1e2bd55ebcdf559e7344cac>

```
use App\Article;
use League\Fractal\Manager;
use Laravelista\Syndra\Syndra;
```

This enables us to use `$this->syndra` and `$this->fractal` from any method inside the class `ArticleController`.

Back to the `index` method. This is what we want to achieve:

- display all published articles sorted by latest
- display all tags for an article
- optionally display an article category

To achieve this there are a few steps:

1. Parse optional data
2. Fetch articles that satisfy the criteria
3. Transform articles using transformers to get desired output
4. Return appropriate response with transformed data

Let's start by adding a dependency to our `index` method so that its declaration looks like so:

```
public function index(Request $request)
```

Now let's add code to tell Fractal to parse optional data if any:

```
// enable parse includes for optional data
if ($request->has('include')) {
    $this->fractal->parseIncludes($request->get('include'));
}
```

We will now fetch all published articles and sort them by the latest.

```
$articles = $this->articles
    ->published()
    ->latest()
    ->get();
```

As you may have noticed now, we are using a scope `Published` to fetch only the published articles but we haven't defined it on our `Article` model yet. We will do that now.

Go to `app/Article.php` and add a method called `scopePublished`:


```
public function scopePublished($query)
{
    return $query->where('published', 1);
}
```

Think of scopes like handy shortcuts that enable you to get cleaner code. Find out more [here](#)²¹⁰.

Transformers

Now that we have fetched the articles we want, it is time to transform them using a transformer class from Fractal.

Let's create a place where we will store all of our transformer classes. In `app/` create a new folder called `Transformers`.

Inside that folder create a file `ArticleTransformer.php` and place the following code:

```
<?php namespace App\Transformers;

use App\Article;
use League\Fractal\TransformerAbstract;

class ArticleTransformer extends TransformerAbstract
{
    public function transform(Article $article)
    {
        return [
        ];
    }
}
```

This is the bare minimum that the transformer class needs. The `transform` method accepts an article and returns an array. Now we have to define what we want our article output to look like:

²¹⁰<https://laravel.com/docs/5.2/eloquent#query-scopes>

```
return [
    'id' => (int) $article->id, // not recommended
    'heading' => $article->heading,
    'content' => $article->content,
    'published' => (boolean) $article->published,
    'published_at' => $article->published_at->format('d.m.Y')
];
```

Usually, you would not want to expose your database record id to the world as it can be a potential security risk. You would use a slug or some other sort of unique identifier instead.

By default, we want to return all tags that the article has, so we have to define that in the transformer:

```
protected $defaultIncludes = [
    'tags',
];
```

now we have to create a method that tells the transformer how to include those tags:

```
public function includeTags(Article $article)
{
    $tags = $article->tags()->get();

    return $this->collection($tags, new TagTransformer);
}
```

As you guessed, we now have to create a transformer class for tags called TagTransformer. Create a new file in app/Transformers called TagTransformer.php and place the following code inside:

```
<?php namespace App\Transformers;

use App\Tag;
use League\Fractal\TransformerAbstract;

class TagTransformer extends TransformerAbstract
{

    public function transform(Tag $tag)
    {
        return [
            'name' => $tag->name,
        ];
    }
}
```

```

    }

}

```

Let's get back to our `ArticleTransformer` class. We said that we wanted to let our API users to optionally get the article category if wanted. To enable that behavior we need to define optional includes:

```

protected $availableIncludes = [
    'category',
];

```

As with the default includes, optional includes also need a method to tell the article transformer class how to include them. Let's do that now.

```

public function includeCategory(Article $article)
{
    $category = $article->category()->first();

    return $this->item($category, new CategoryTransformer);
}

```

Because an article can only belong to one category, we use `Fractal::itemResource` and not `collection` as we used with tags. We still need to create a transformer for `Category`.

Create a new file in `app/Transformers` called `CategoryTransformer.php` and place the following code inside:

```

<?php namespace App\Transformers;

use App\Category;
use League\Fractal\TransformerAbstract;

class CategoryTransformer extends TransformerAbstract
{
    public function transform(Category $category)
    {
        return [
            'name' => $category->name,
        ];
    }
}

```

Both Category and Tag models could have many other fields except name™, but for this tutorial I simplified it.

We have now completed all transformers that we will need to transform articles. Let's get back to our ArticleController in index method and add the code responsible for transforming articles using Fractal.

```
$resource = new Collection($articles, new ArticleTransformer);
$data = $this->fractal->createData($resource)->toArray();
```

Don't forget to add *use statements* at the top of the ArticleController just below the namespace declaration:

```
use League\Fractal\Resource\Collection;
use App\Transformers\ArticleTransformer;
```

Finally, we have to return the response and verify that it works as we expect it to work.

Add the following code at the bottom of the index method:

```
return $this->syndra->setHeaders([
    'Access-Control-Allow-Origin' => '*',
])->respond($data);
```

To enable your API to be used by others on the Internet we set a header that allows that. See [here](https://www.w3.org/wiki/CORS_Enabled)²¹¹ for more details.

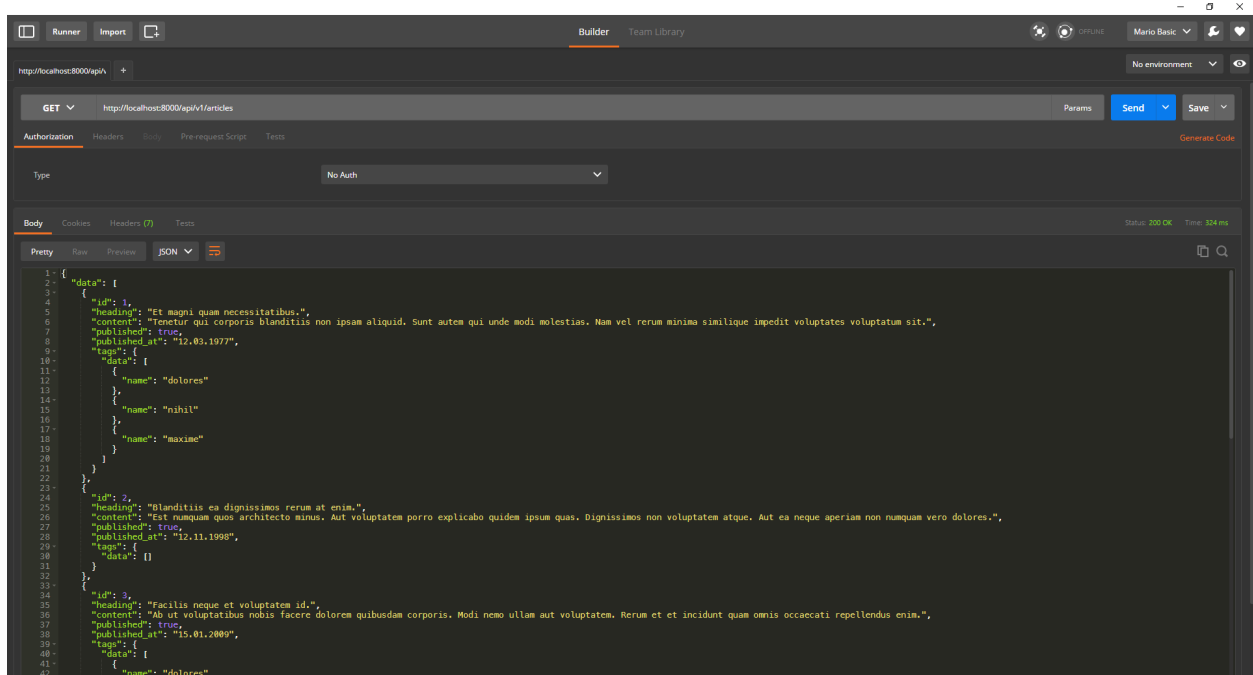
If you want to clean your code even further, you could tell Syndra to set the headers in the constructor of ArticleController, so that you don't have to set it in every method of the controller.

To test that everything works at this point in our development we can use the browser or the Postman application I mentioned before. *I suggest using Postman* but do as you wish for now :)

To run the application I am using `php artisan serve` and in my case, it is running on `http://localhost:8000`.

To view all articles using our API you need to point your browser or Postman to `http://localhost:8000/api/v1/articles`.

²¹¹https://www.w3.org/wiki/CORS_Enabled



Postman Index

This is just a sample of how the output should look like:

```

{
  "data": [
    {
      "id": 1,
      "heading": "Et magni quam necessitatibus.",
      "content": "Tenetur qui corporis blanditiis non ipsam aliquid. Sunt autem \
qui unde modi molestias. Nam vel rerum minima similique impedit voluptates volup\
tatum sit.",
      "published": true,
      "published_at": "12.03.1977",
      "tags": {
        "data": [
          {
            "name": "dolores"
          },
          {
            "name": "nihil"
          },
          {
            "name": "maxime"
          }
        ]
      }
    }
  ]
}

```

```

    ]
  }
},
{
  "id": 2,
  "heading": "Blanditiis ea dignissimos rerum at enim.",
  "content": "Est numquam quos architecto minus. Aut voluptatem porro explic\
abo quidem ipsum quas. Dignissimos non voluptatem atque. Aut ea neque aperiam no\
n numquam vero dolores.",
  "published": true,
  "published_at": "12.11.1998",
  "tags": {
    "data": []
  }
}
]
}

```

By default tags are included in the response, but if you remember we also made it possible to display article category using optional includes. To test that that is also working we use this URL <http://localhost:8000/api/v1/articles?include=category>.

The index method is now complete.

This is the current state of the repository at this moment [83c767b6a248f23a5c9036c205f2cc43fc6faf2c](https://github.com/laravelista/laravel-api-101-build/tree/83c767b6a248f23a5c9036c205f2cc43fc6faf2c)²¹².

Show

Once you have done the `index` method, the `show` method is a walk in the park. One thing to watch for in this method is what if the requested model does not exist. I will show you how to handle that situation using `Syndra`.

The `show` method contains almost the same steps as the `index` method:

To achieve this there are a few steps:

1. Parse optional data
2. Fetch single article by id that satisfies the criteria
3. Transform article using transformers to get desired output
4. Return appropriate response with transformed data

Let's start by adding a dependency to our `show` method so that its declaration looks like:

²¹²<https://github.com/laravelista/laravel-api-101-build/tree/83c767b6a248f23a5c9036c205f2cc43fc6faf2c>

```
public function show($id, Request $request)
```

Now we parse optional includes:

```
if ($request->has('include')) {
    $this->fractal->parseIncludes($request->get('include'));
}
```

Find published article by id or fail.

```
$article = $this->articles
    ->published()
    ->findOrFail($id);
```

Transform article using Fractal ArticleTransformer we created during the index method.

```
$resource = new Item($article, new ArticleTransformer);
$data = $this->fractal->createData($resource)->toArray();
```

Don't forget to add the *use* statement for Item at the top of the file just below the namespace definition:

```
use League\Fractal\Resource\Item;
```

And finally we return the transformed data using Syndra:

```
return $this->syndra->setHeaders([
    'Access-Control-Allow-Origin' => '*',
])->respond($data);
```

Handle model not found exception

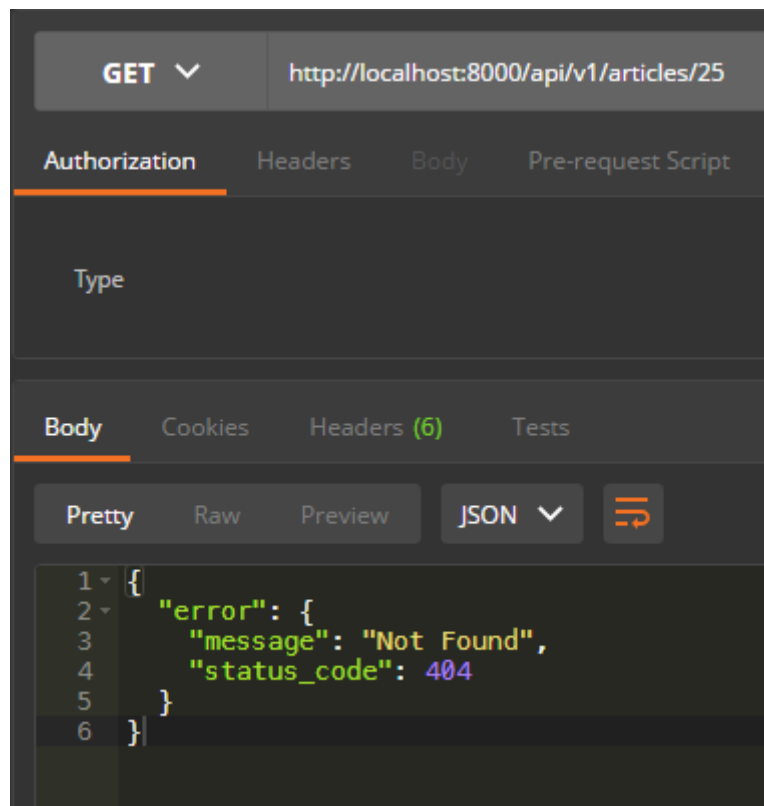
This completes the show method, but what about a situation if the model for given id is not found. In case a `ModelNotFoundException` is thrown we have to intercept it and return an appropriate response.

Go to `app/Exceptions/Handler.php` and before the return statement in the `render` method add this code:

```
if ($e instanceof ModelNotFoundException) {
    return \Syndra::respondNotFound();
}
```

So now, every time a `ModelNotFoundException` is thrown in the application this code will respond with `Syndra::respondNotFound` method and the output will look like this:

```
{
  "error": {
    "message": "Not Found",
    "status_code": 404
  }
}
```



Postman Show

This is the current state of the repository at this moment [64d95a4af248cc75fb87124aa561bf976ace712a](https://github.com/laravelista/laravel-api-101-build/tree/64d95a4af248cc75fb87124aa561bf976ace712a)²¹³.

²¹³<https://github.com/laravelista/laravel-api-101-build/tree/64d95a4af248cc75fb87124aa561bf976ace712a>

Store

In the store method, we want to create/store a new article using our API in the database. This is the workflow that we want:

1. Pass required data to create a new article
2. Validate given data
3. Create new article
4. Get appropriate response in return

Disable CSRF protection for API routes

Before we proceed we need to disable CSRF protection for API routes. We do that by going to `app/Http/Middleware/VerifyCsrfToken.php` file and adding a URI `api/*` to the `except` property so that it looks like this:

```
protected $except = [
    'api/*'
];
```

To find out more about CSRF protection [click here](#)²¹⁴.

Custom AJAX validation error response

The default Laravel behavior is this “When using the `validate` method during an AJAX request, Laravel will not generate a redirect response. Instead, Laravel generates a JSON response containing all of the validation errors. This JSON response will be sent with a 422 HTTP status code.” - taken from [official Laravel documentation](#)²¹⁵.

If you want, you are free to leave the default Laravel response, but if you want to replace it with Syndra’s response read on.

Go to `app/Exceptions/Handler.php` and in `render` method just below the `ModelNotFoundException` code we added there, add this block of code:

²¹⁴<https://laravel.com/docs/5.2/routing#csrf-protection>

²¹⁵<https://laravel.com/docs/5.2/validation#quick-ajax-requests-and-validation>

```
if ($e instanceof ValidationException) {
    return \Syndra::respondValidationError(
        $e->validator->errors()->getMessages()
    );
}
```

This block of code checks if `ValidationException` is thrown and then responds with `Syndra::respondValidationError`. All error messages are displayed in the response.

This is what the response would look like:

```
{
  "error": {
    "message": {
      "heading": [
        "The heading field is required."
      ],
      "content": [
        "The content field is required."
      ],
      "category_id": [
        "The category id field is required."
      ]
    },
    "status_code": 422
  }
}
```

Let's add validation to the store method now:

```
$this->validate($request, [
    'heading' => 'required|string',
    'content' => 'required|string',
    'published' => 'boolean',
    'published_at' => 'date',
    'category_id' => 'bail|required|integer|exists:categories,id'
]);
```

The required fields are:

- *heading* - string
- *content* - text

- *category_id* - this needs to be an existing category id

Optional fields are:

- *published* - boolean
- *published_at* - valid date 2016-05-20 00:00:00

We could have created a “form request” to handle this validation for us so that we keep the controller clean. To find out more about this [click here](#)²¹⁶.

This is the current state of the repository at this moment [c7e8aa6b50e85013caa6ae119af7ea95ed654b96](#)²¹⁷.

Now we will create a new article with our validated data. But remember, the article must have a *category_id* field before it can be saved to the database.

Article model must be filled with attributes, but not saved before the category is associated. To do that we will use `fill` method instead of `create` of the model.

```
$article = $this->articles->fill([
    'heading' => $request->get('heading'),
    'content' => $request->get('content'),
    'published' => $request->get('published', false),
    'published_at' => $request->get('published_at')
]);
```

If the request does not contain published data it will default to false, but we could have left it blank because in our migration we have set the default value for published to be false. *published_at* can be null because we defined it as nullable in the migration.

Now we have to find the category by given *category_id* and associate it with this new article.

```
$category = \App\Category::find($request->get('category_id'));
$article->category()->associate($category);
$article->save();
```

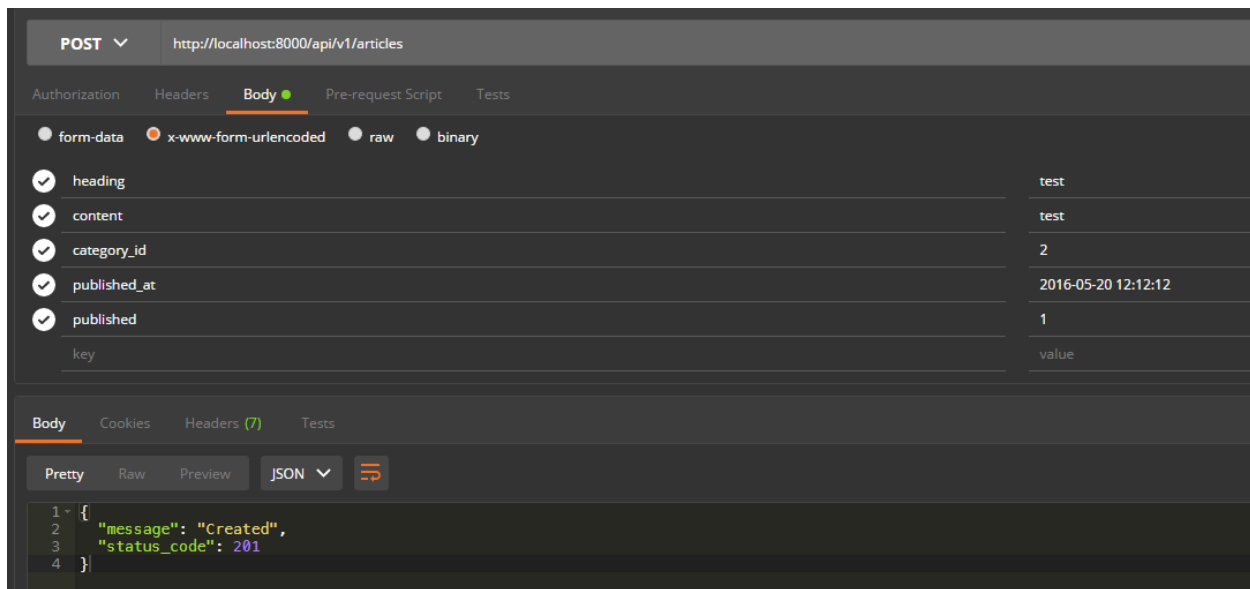
Finally we save the article model to database and return `Syndra::respondCreated` response:

²¹⁶ <https://laravel.com/docs/5.2/validation#form-request-validation>

²¹⁷ <https://github.com/laravelista/laravel-api-101-build/tree/c7e8aa6b50e85013caa6ae119af7ea95ed654b96>

```
return $this->syndra->setHeaders([
    'Access-Control-Allow-Origin' => '*',
])->respondCreated();
```

The store method is now complete, you can test that it works by using Postman, and I will show you how now.



Postman store

Set *method* to be POST and the URL should point to `api/v1/articles` as it was for our `index` method. Then click on *body* tab and select `x-www-form-urlencoded`. There set the data you want to send to the store method. Remember that `heading`, `content` and `category_id` are required. Click on *Send* and if the data entered is valid you will get a response like this:

```
{
  "message": "Created",
  "status_code": 201
}
```

You can verify that the model has been created either by using the article `index` endpoint, using `php artisan tinker` or querying the database directly.

Remember that if you did not set `published` to be 1, the article will not show in `index` endpoint because the `index` endpoint shows only published articles.

This is the current state of the repository at this moment [1826ff195e46b7e6de349edecdedb4ff9d07f6a0](https://github.com/laravelista/laravel-api-101-build/tree/1826ff195e46b7e6de349edecdedb4ff9d07f6a0)²¹⁸.

²¹⁸<https://github.com/laravelista/laravel-api-101-build/tree/1826ff195e46b7e6de349edecdedb4ff9d07f6a0>

Update

The update method is very similar to the store method. I will go step by step so that you can follow along.

The validation has no required fields this time:

```
$this->validate($request, [
    'heading' => 'string',
    'content' => 'string',
    'published' => 'boolean',
    'published_at' => 'date',
    'category_id' => 'bail|integer|exists:categories,id'
]);
```

We now have to find the article by given id:

```
$article = $this->articles->findOrFail($id);
```

Then we update the fields:

```
$article->update([
    'heading' => $request->get('heading', $article->heading),
    'content' => $request->get('content', $article->content),
    'published' => $request->get('published', $article->published),
    'published_at' => $request->get('published_at', $article->published_at)
]);
```

If the category_id is passed, we need to update that also:

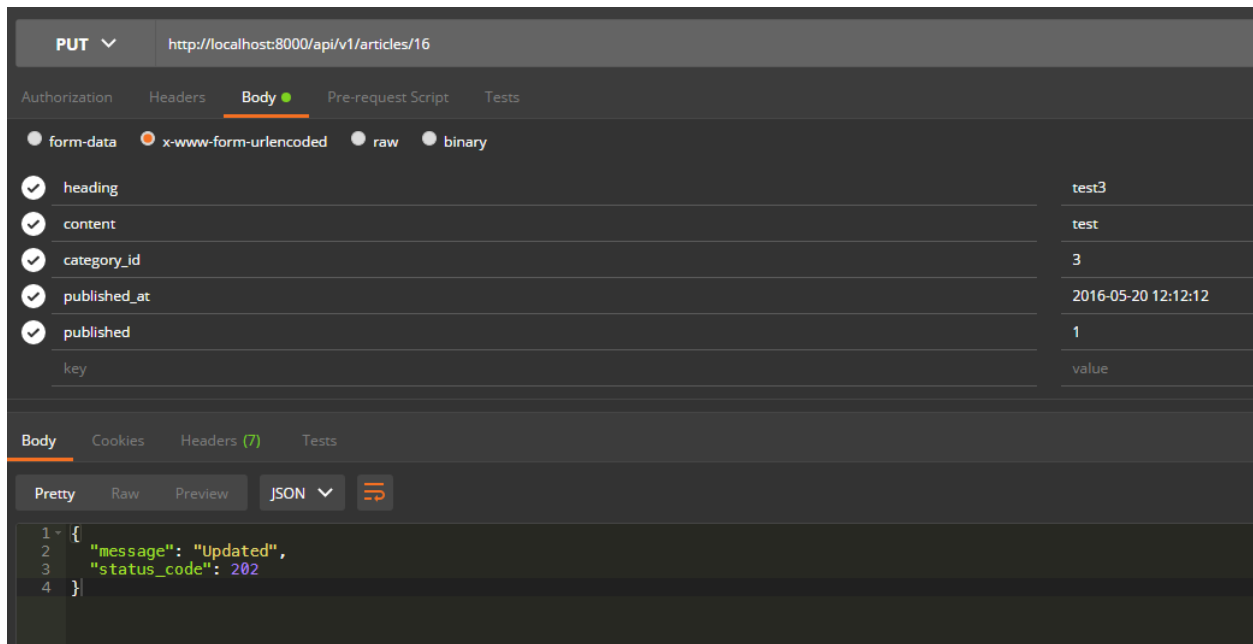
```
if($request->has('category_id')) {
    $category = \App\Category::find($request->get('category_id'));
    $article->category()->associate($category);
    $article->save();
}
```

And finally we return Syndra::respondUpdated response:

```
return $this->syndra->setHeaders([
    'Access-Control-Allow-Origin' => '*',
])->respondUpdated();
```

This is the current state of the repository at this moment [dbc74ee5bf22130d8a730657a626f712f96e1116](https://github.com/laravelista/laravel-api-101-build/tree/dbc74ee5bf22130d8a730657a626f712f96e1116)²¹⁹.

Now to test that everything works as expected.



Postman update

Set *method* to be PUT and the URL should point to `api/v1/articles/{id}` (replace `{id}` with your actual article id from database). Then click on *body* tab and select `x-www-form-urlencoded`. There set the data you want to send to the store method. Remember that there are no required fields, so set only the fields that you want to change. Click on *Send* and if the data entered is valid you will get a response like this:

```
{
  "message": "Updated",
  "status_code": 202
}
```

Destroy

This is very simple. We just need to find the article by `id` and then delete it.

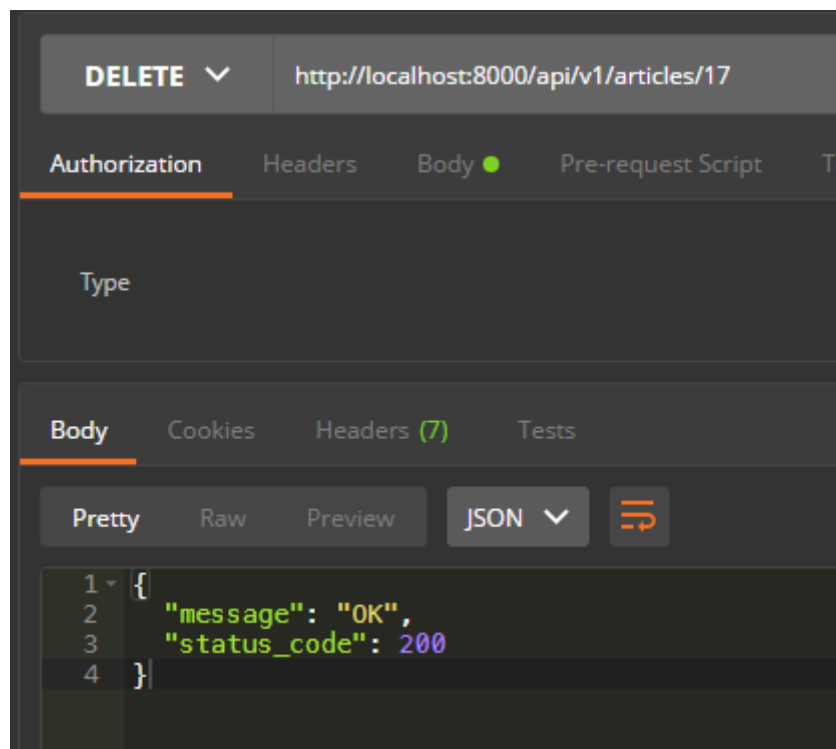
²¹⁹ <https://github.com/laravelista/laravel-api-101-build/tree/dbc74ee5bf22130d8a730657a626f712f96e1116>

```
$article = $this->articles->findOrFail($id);
$article->delete();
```

We will return a standard 200 response that everything is OK.

```
return $this->syndra->setHeaders([
    'Access-Control-Allow-Origin' => '*',
])->respondOk();
```

Let's test that it works using Postman:



Postman destroy

Set *method* to be DELETE and the URL should point to `api/v1/articles/{id}` (replace {id} with your actual article id from the database that you want to delete). Click on *Send* and if the article is found, will get a response like this:

```
{
  "message": "OK",
  "status_code": 200
}
```

This is the current state of the repository at this moment [e5a0b2ac8f08c01704051d638966a7978b25e4a0](https://github.com/laravelista/laravel-api-101-build/tree/e5a0b2ac8f08c01704051d638966a7978b25e4a0)²²⁰.

Now we have completed the whole Article controller and it is fully working. You can:

- view all articles
- view a single article
- create a new article
- update an existing article
- delete an article

You now have full CRUD functionality for articles. You can apply the same logic to create Tag and Category endpoints for this API.

One thing that we are missing at this point is an **authentication system**. Currently, everyone can use the API however they want and do whatever they want (if you were to put this application on the Internet right now, you would have a public API that everyone can use). This is generally not a good idea.

In the next part of this course, I will show you how to make use of the Laravel Token Based Authentication system.

AUTH

I will show you how to implement native Laravel Token Based Authentication in our API.

Published at: 23. May, 2016.



View Source Code

Source code for this tutorial is available [here](https://github.com/laravelista/laravel-api-101-build)²²¹.

Now that we have built our API in the previous tutorial, we need a way to protect it, so that only authenticated users can use it. We will do that by using Token Based Authentication that comes with Laravel by default.

Laravel comes with built-in token-based authentication, but the documentation on it is almost nonexistent. After I complete this tutorial, I will write the documentation for it and send a [PR](https://github.com/laravel/docs/pull/2325)²²² to Laravel so that it gets included in the official documentation.

²²⁰<https://github.com/laravelista/laravel-api-101-build/tree/e5a0b2ac8f08c01704051d638966a7978b25e4a0>

²²¹<https://github.com/laravelista/laravel-api-101-build>

²²²<https://github.com/laravel/docs/pull/2325>

Using a native authentication system is pretty handy because you don't have to deal with integrating it with your framework, it just works.

In short, we will give each user a token, with which he can access the API and protect the routes that we want to require authentication. I will also show you three different ways of sending the token to the API for authentication.

Preparations

There are some things that we can do to make our API faster.

Warning! If you are supplementing your existing application with an API section, you can skip this part.

These preparations are only useful if you are building an *API only* application because we are going to discard some things that an API normally does not need.

Change default route middleware

By default all routes found in `app/Http/routes.php` are assigned to web middleware group. web middleware group by default comes with these middlewares:

- EncryptCookies
- AddQueuedCookiesToResponse
- StartSession
- ShareErrorsFromSession
- VerifyCsrfToken

All those middlewares are not required for our API, so it would be nice to stop using them. Less middleware to hop through equals faster code. What we want to actually do is to switch routes found in `app/Http/routes.php` to api middleware group which by default comes with API throttling (60 requests in 1 minute).

To change this default behavior go to `app/Providers/RouteServiceProvider.php` and find a method called `mapWebRoutes`. There locate a line where it says:

```
'namespace' => $this->namespace, 'middleware' => 'web',
```

and change it to:

```
'namespace' => $this->namespace, 'middleware' => 'api',
```

To test that are changes are working from the command line run `php artisan route:list`:

```
$ php artisan route:list
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Domain | Method | URI | Name | Middleware | Action |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|        | GET|HEAD | api/v1/articles | api.v1.articles.index | Api |
p\Http\Controllers\ArticleController@index | api |
|        | POST | api/v1/articles | api.v1.articles.store | Api |
p\Http\Controllers\ArticleController@store | api |
|        | GET|HEAD | api/v1/articles/{articles} | api.v1.articles.show | Api |
p\Http\Controllers\ArticleController@show | api |
|        | PUT|PATCH | api/v1/articles/{articles} | api.v1.articles.update | Api |
p\Http\Controllers\ArticleController@update | api |
|        | DELETE | api/v1/articles/{articles} | api.v1.articles.destroy | Api |
p\Http\Controllers\ArticleController@destroy | api |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|        |        |        |        |        |        |
```

You should see that the middleware column now says `api` instead of `web` as it did before.

Change authentication defaults

By default, Laravel uses a web guard to authenticate requests. What we want to do is to make it use the api guard. To do that go to `config/auth.php` and under *Authentication Defaults* section change default guard to `api` like so:

```
'defaults' => [
    'guard' => 'api',
    'passwords' => 'users',
],
```

This is the current state of the repository at this moment [0cf7064ea399b7473a6c611ee1ea2a9135fd90bb](https://github.com/laravelista/laravel-api-101-build/tree/0cf7064ea399b7473a6c611ee1ea2a9135fd90bb)²²³.

Token Based Authentication

As mentioned before, the official Laravel documentation for Token-based authentication is very slim, almost nonexistent. I will make an effort to change that in the future.

In token-based authentication, you assign each user a token which he then sends in each request to the API in order to authenticate.

²²³<https://github.com/laravelista/laravel-api-101-build/tree/0cf7064ea399b7473a6c611ee1ea2a9135fd90bb>

Users API token migration

I don't know if you have noticed, but when you first migrated the database for this tutorial, it already came with `users` and `password_resets` tables. Those are the migrations that come out of the box with Laravel and they are ready for normal authentication.

To support Token Based Authentication (in future reference TBA) we need to add a column `api_token` to the `users` table. We will make a new migration and in it add that column.

To create a new migration from the command line run:

```
php artisan make:migration add_api_token_to_users --table=users
```

Find the migration file in `database/migrations/` and make the `up` method look like this:

```
Schema::table('users', function (Blueprint $table) {
    $table->string('api_token', 60)->nullable()->unique();
});
```

The length of the field `api_token` can be changed depending on your token generation way.

Also, we need to make the reverse of what we did here in the `down` method, so change the `down` method to look like this:

```
Schema::table('users', function (Blueprint $table) {
    $table->dropColumn('api_token');
});
```

Run the migration with:

```
php artisan migrate
```

Now let's add `api_token` field to the list of mass assignable fields in `app/User.php`. Add `api_token` to `fillable` property on the model:

```
protected $fillable = [
    'name', 'email', 'password', 'api_token',
];
```

Adding `api_token` to the `fillable` list is not required, but it makes sense to me to add it there.

This is the current state of the repository at this moment [a94b99a5c1fe168a4c0fd4a3774cb7be847cbe7c](https://github.com/laravelista/laravel-api-101-build/tree/a94b99a5c1fe168a4c0fd4a3774cb7be847cbe7c)²²⁴.

²²⁴<https://github.com/laravelista/laravel-api-101-build/tree/a94b99a5c1fe168a4c0fd4a3774cb7be847cbe7c>

Seeding a user

Currently, we don't have any users in our database. To do that we will create a seeder class that creates a single user and assigns an `api_token` to him.

To create a seeder class using command line type:

```
php artisan make:seeder UserSeeder
```

Now, let's create a new user and assign a token to him. In `database/seeds/UserSeeder.php` add this code to the `run` method:

```
\App\User::create([
    'name' => 'API Dummy',
    'email' => 'dummy@api.app',
    'password' => bcrypt('password'),
    'api_token' => bin2hex(openssl_random_pseudo_bytes(16))
]);
```

We are using here `bin2hex(openssl_random_pseudo_bytes(16))` to generate a unique token for our API. This generates a 32 character long token. If you want, you can replace that line with `str_random(60)` to generate a 60 character long token. *There are different ways of generating a token, so be sure to replace this line with whatever suits your needs.*

To seed the user to the database we want to call once:

```
$ php artisan db:seed --class=UserSeeder
```

Now we have our first API user.

This is the current state of the repository at this moment [2ac86733bb17bb0f344ea7e0341df67335df912b](https://github.com/laravelista/laravel-api-101-build/tree/2ac86733bb17bb0f344ea7e0341df67335df912b)²²⁵.

Protecting routes

All requirements for TBA are complete. We now need to protect the routes that we want to require authentication. We want to let `index` and `show` endpoints to be public so that everyone can access them, but we want to protect `store`, `update` and `destroy` endpoints so that they require authentication.

To do this go to `app/Http/Controllers/ArticleController.php` and in constructor add the following at the bottom:

²²⁵ <https://github.com/laravelista/laravel-api-101-build/tree/2ac86733bb17bb0f344ea7e0341df67335df912b>

```
$this->middleware('auth:api', ['only' => [
    'store',
    'update',
    'destroy'
]]);
```

This tells article controller to protect only store, update and destroy methods.

If you made the changes to your application as mentioned in the Preparations chapter, you can omit `:api` from `auth:api` declaration, because we already told our application to use the `api` guard by default.

To make sure that this works you can use the command line `php artisan route:list`:

```
$ php artisan route:list
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| Domain | Method | URI | Name | Middleware | Action |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|        | GET|HEAD | api/v1/articles | api.v1.articles.index | api | Ap\
p\Http\Controllers\ArticleController@index |
|        | POST | api/v1/articles | api.v1.articles.store | api,auth:api | Ap\
p\Http\Controllers\ArticleController@store |
|        | GET|HEAD | api/v1/articles/{articles} | api.v1.articles.show | api | Ap\
p\Http\Controllers\ArticleController@show |
|        | PUT|PATCH | api/v1/articles/{articles} | api.v1.articles.update | api,auth:api | Ap\
p\Http\Controllers\ArticleController@update |
|        | DELETE | api/v1/articles/{articles} | api.v1.articles.destroy | api,auth:api | Ap\
p\Http\Controllers\ArticleController@destroy |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
|        | GET|HEAD | api/v1/articles | api.v1.articles.index | api | Ap\
p\Http\Controllers\ArticleController@index |
|        | POST | api/v1/articles | api.v1.articles.store | api,auth:api | Ap\
p\Http\Controllers\ArticleController@store |
|        | GET|HEAD | api/v1/articles/{articles} | api.v1.articles.show | api | Ap\
p\Http\Controllers\ArticleController@show |
|        | PUT|PATCH | api/v1/articles/{articles} | api.v1.articles.update | api,auth:api | Ap\
p\Http\Controllers\ArticleController@update |
|        | DELETE | api/v1/articles/{articles} | api.v1.articles.destroy | api,auth:api | Ap\
p\Http\Controllers\ArticleController@destroy |
```

Or you can use Postman to create a new article as described in the previous tutorial. *You will get unauthorized JSON response.*

Protecting all routes

This step is optional. In some cases you want to protect the whole API, to make it private.

To do that go to `app/Http/routes.php` and add `'middleware' => 'auth:api'` to our API route group like so:

```
Route::group(['prefix' => 'api/v1', 'middleware' => 'auth:api'], function ()
{
    Route::resource('articles', 'ArticleController', ['except' => ['create', 'edit\
it']]);
});
```

Now all routes inside that group require authentication.

Handling unauthorized requests

Before testing that this above code works we need to handle a case if the request fails authentication. In current state, if the authentication fails it will try to redirect you to the “login” page or return a response 401 Unauthorized (This barely ever happens).

To change what happens when the authentication fails go to `app/Http/Middleware/Authenticate.php` and make the `handle` method look like this:

```
if (Auth::guard($guard)->guest()) {
    return \Syndra::respondUnauthorized();
}

return $next($request);
```

Now every time authentication fails, the user will be given a JSON response with 401 status code and appropriate message:

```
{
  "error": {
    "message": "Unauthorized",
    "status_code": 401
  }
}
```

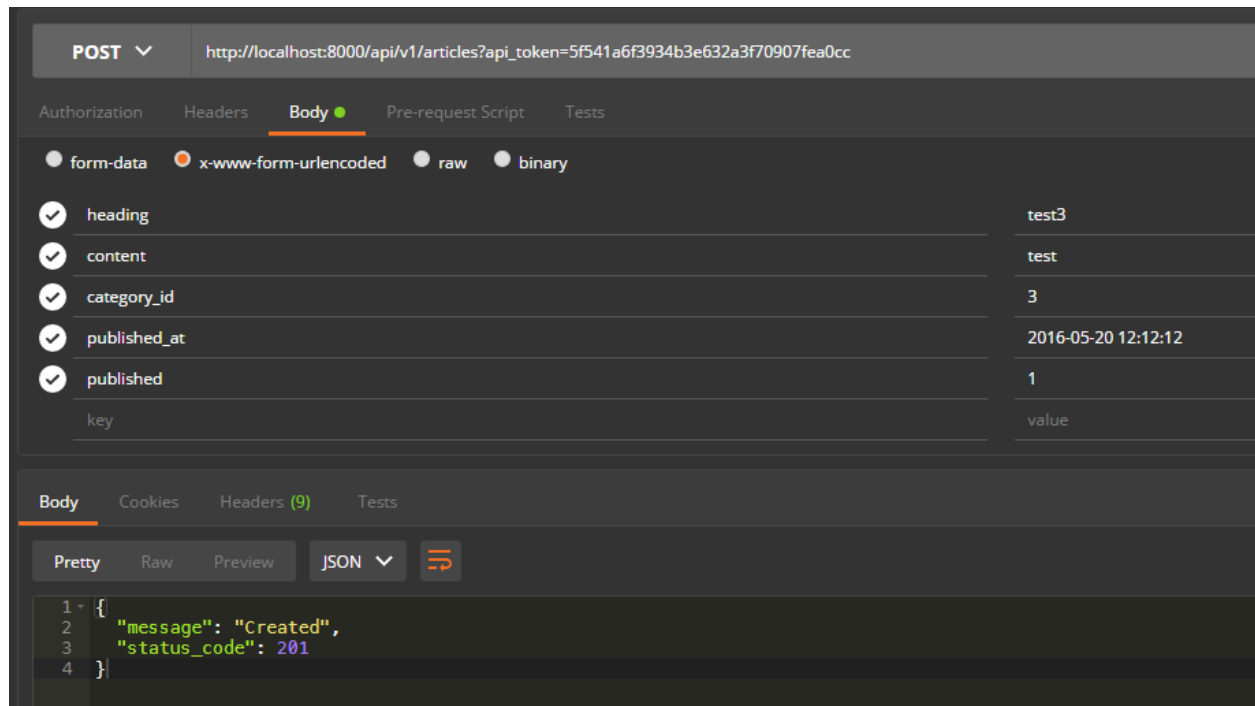
This is the current state of the repository at this moment [4ed699768c6662fa67edd2ef644f1620a6239d25](https://github.com/laravelista/laravel-api-101-build/tree/4ed699768c6662fa67edd2ef644f1620a6239d25)²²⁶.

Accessing protected routes

I will now show you three ways of authenticating a user to our API to store, update or destroy an article. The examples given here will be for the store endpoint, but you can apply the same principles for other endpoints.

²²⁶<https://github.com/laravelista/laravel-api-101-build/tree/4ed699768c6662fa67edd2ef644f1620a6239d25>

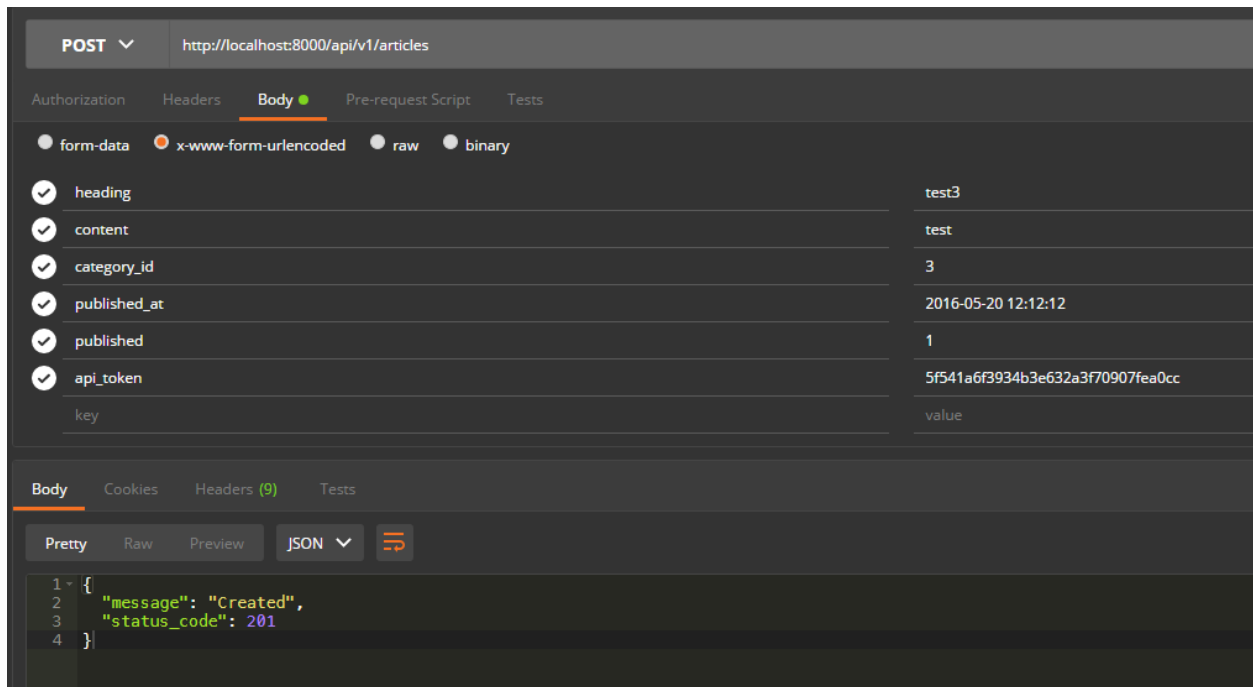
Query parameter



Query parameter

This one is the simplest. You just append `?api_token=<your_token_here>` to the end of the URL `POST /api/v1/articles`.

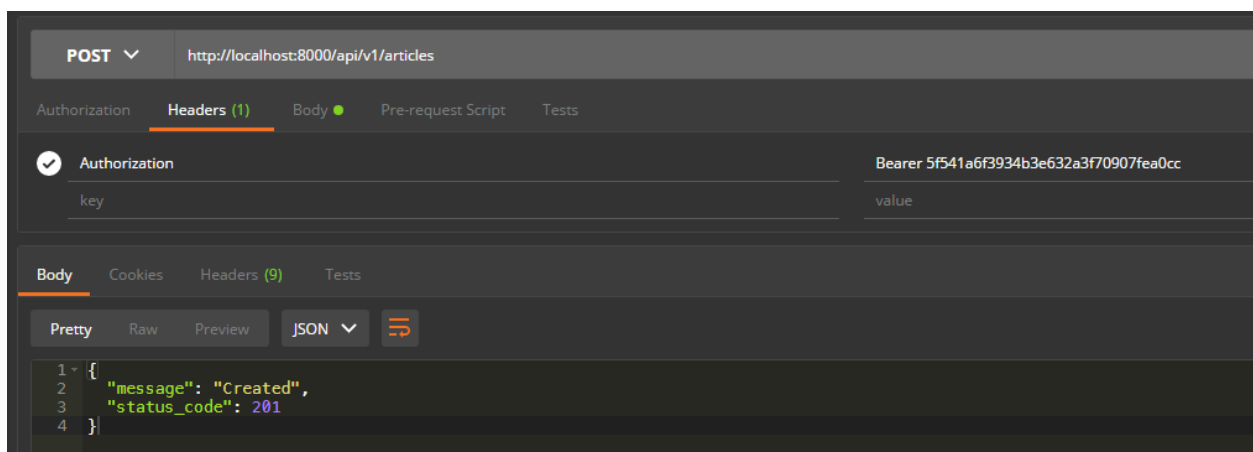
Request body



Request body

Another easy one. To the body of the request, along with other data like heading and content, you pass `api_token` with the value of your token. Be sure to use `x-www-form-urlencoded` in Postman.

Header Authorization Bearer



Header Bearer

This is my personal recommendation and this one I use in my own projects. You need to add a header to the request for `Authorization:Bearer <your_token_here>`.

In Postman click on *Headers* tab and start typing in *key* field Authorization, a drop-down will show so that you can select it. In *value* field type Bearer <your_token_here>. Be sure to replace <your_token_here> with the actual token that you got from seeding the user.

You now have Token Based Authentication ready for your API and now you can now test different ways of authenticating to it.

In the next part of this course, I will show you how to consume the API we just built.

CONSUME

We will create a new Laravel application which connects to the API that we have built in the previous tutorials.

Published at: 12. December, 2016.



View Source Code

Source code for this tutorial is available [here](#)²²⁷.

It's alive! In this tutorial, we will create a new Laravel 5.3 application which will use Guzzle to connect to our API that we have built in the previous tutorials. We will fetch the articles, display them on the homepage and provide links to view each article.

We will create a service class which will handle retrieving the articles from the API. In return, we will get a Collection with which we can do whatever we want. We will be using the `index` and `show` methods from our API to retrieve articles.

Why this approach?

If you have a single source of truth, in this case, the articles database, and you want to display them on multiple sites, it is much easier to retrieve them from the API than to duplicate models on other sites and connect directly to the database. If you duplicate models, you will have to maintain them, and trust me this quickly becomes a pain in the ass.

With this approach, you will get a collection of articles with which you can do whatever you want. If you use this approach then it does not matter to you, which backend technology is behind the API. You can move your API to Python, Node.js... it does not matter since you always get the same response from the API. Same applies to the sites where you present the resources from the API. It can be a Laravel application, Django, JavaScript or whatever.

You can change the technology and code behind the API, but as long as it returns the same formatted responses, all other sites that depend on it will continue to work.

This way it is much easier to maintain applications since we have decoupled the technologies behind them.

²²⁷ <https://github.com/laravelista/laravel-api-101-consume>

Quick Recap

In the previous tutorials, we have created the API and implemented an authentication system using token based authentication.

API installation

You can skip this if you already have a working API built in the previous tutorials.

Clone the repository and navigate inside it:

```
git clone git@github.com:laravelista/laravel-api-101-build.git
```

```
cd laravel-api-101-build
```

Install dependencies:

```
composer install
```

Copy file `.env.example` to `.env` file:

```
cp .env.example .env
```

and change the `APP_KEY` in `.env` using:

```
php artisan key:generate
```

Create a empty database file called `database.sqlite` in database directory.

Run migrations and seed database with:

```
php artisan migrate --seed
```

To start the API run:

```
php artisan serve
```

Your API is now available on `http://localhost:8000`.

To recap, here are the endpoints that are available on the API:

View all articles

Endpoint: GET `api/v1/articles`.

Available includes:

- Category GET `api/v1/articles?include=category`

View a single article

Endpoint: GET `api/v1/articles/{id}`.

Available includes:

- Category GET `api/v1/articles?include=category`

Create a single article

Endpoint: POST `api/v1/articles`.

This endpoint is protected and requires authentication to work. See chapter *Authentication*.

The required fields are:

- heading - string
- content - text
- category_id - this needs to be an existing category id. See chapter *Category*.

Optional fields are:

- published - boolean
- published_at - valid date 2016-05-20 00:00:00

Update an existing article

Endpoint: PUT `api/v1/articles/{id}`.

This endpoint is protected and requires authentication to work. See chapter *Authentication*.

All fields are optional.

Delete an article

Endpoint: DELETE `api/v1/articles/{id}`.

This endpoint is protected and requires authentication to work. See chapter *Authentication*.

Authentication

To get the `api_token` for our dummy user, from the command line do:

```
php artisan tinker
```

```
App\User::first()->api_token;
```

```
cafbef5071387b4e4646732334dddf03
```

Your token will be different. Write it down somewhere, because you will need it.

Accessing protected routes

There are three ways to pass the `api_token`. Using the *query parameter*, *request body* or *header authorization Bearer*

Query Parameter

Append `?api_token=<your_token_here>` to the end of the URL `POST /api/v1/articles`.

Request Body

To the body of the request, along with other data like heading and content, you pass `api_token` with the value of your token. Be sure to use `x-www-form-urlencoded` in Postman.

Header Authorization Bearer

You need to add a header to the request for `Authorization:Bearer <your_token_here>`.

Category

Since we haven't implemented Category endpoints for our API, we have to manually retrieve a category id.

From the command line:

```
php artisan tinker
```

```
App\Category::first()->id;
```

```
1
```

Or you can create a new category:

```
php artisan tinker
```

```
App\Category::create(['name' => 'News']);
```

You will see a category id, you will need it to create a new article.

Postman

I have created a collection in [Postman](https://www.getpostman.com/collections/1111111)²²⁸ so that you can easily interact with the API: <https://www.getpostman.com/collections/1111111> once you run it locally.

Getting started

I have created an empty Laravel 5.3 application repository on GitHub. You can view it [here](https://github.com/laravelista/laravel-api-101-consume)²²⁹. Now, I will quickly create everything we need, before we continue to creating the service class for consuming our API.

We will need routes for all articles and single articles. One controller to handle those routes. A layout page and two views. We will leverage [Bootstrap](http://getbootstrap.com/)²³⁰ here to quickly create a basic layout and design for the site.

Routes

We will only need two routes. Go to `routes/web.php` and replace the content of the file, so that it looks like this:

```
<?php
Route::get('/', ['uses' => 'ArticleController@index', 'as' => 'home']);
Route::get('articles/{article}', ['uses' => 'ArticleController@show', 'as' => 'articles.show']);
```

Controllers

Let's make an Article Controller now using artisan:

```
php artisan make:controller ArticleController
```

Open the file `app/Http/Controllers/ArticleController.php` and add these methods inside:

²²⁸ <https://www.getpostman.com/collections/1111111>

²²⁹ <https://github.com/laravelista/laravel-api-101-consume>

²³⁰ <http://getbootstrap.com/>

```

public function index()
{
    return view('articles.index');
}

public function show()
{
    return view('articles.show');
}

```

The next step is to create view files `articles/index.blade.php` and `articles/show.blade.php`, but before we do that we will create a master layout page and include Bootstrap in it.

Layout & Views

Layout

Create a new file in `resources/views` called `layout.blade.php` and paste the following inside:

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>@yield('title', 'Laravel API 101 - CONSUME')</title>

    <!-- Latest compiled and minified CSS -->
    <link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7\
/css/bootstrap.min.css" integrity="sha384-BVYiiSIFeK1dGmJRAKycuHAHRg320mUcww7on3\
RYdg4Va+PmSTsz/K68vbdEjh4u" crossorigin="anonymous">
  </head>
  <body>

    <nav class="navbar navbar-default">
      <div class="container">
        <div class="navbar-header">
          <button type="button" class="navbar-toggle collapsed" data-toggle="col\
lapse" data-target="#navbar" aria-expanded="false" aria-controls="navbar">
            <span class="sr-only">Toggle navigation</span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
          </div>

```

```

        </button>
        <a class="navbar-brand" href="{{ route('home') }}">Laravel API 101</a>
    </div>
    <div id="navbar" class="navbar-collapse collapse">
        <ul class="nav navbar-nav">
            <li><a href="{{ route('home') }}">Home</a></li>
        </ul>
    </div><!--/.navbar-collapse -->
</div>
</nav>

@yield('content')

<!-- jQuery (necessary for Bootstrap's JavaScript plugins) -->
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.12.4/jquery.min.\
js"></script>
<!-- Latest compiled and minified JavaScript -->
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.mi\
n.js" integrity="sha384-Tc5IQib027qvyjSMfHjOMaLkfuWVxZxUPnCJA712mCWNIpG9mGCD8wGN\
IcPD7Txa" crossorigin="anonymous"></script>
</body>
</html>

```

Now, let's create the views:

Homepage

Create a folder called articles in resources/views.

Inside the articles folder create a new file called index.blade.php. Paste the following code inside:

```

@extends('layout')

@section('title', 'Home page')

@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <h1>All Articles</h1>
            </div>
        </div>
    </div>
@stop

```

Article Page

Inside the resources/views/articles folder create a new file called show.blade.php. Paste the following code inside:

```
@extends('layout')

@section('title', 'Article Page')

@section('content')
    <div class="container">
        <div class="row">
            <div class="col-md-12">
                <h1>Single Articles</h1>
            </div>
        </div>
    </div>
@stop
```

That should do it. You can now visit <http://localhost:8888> to view the Home page and <http://localhost:8888/articles/xy> to view the Article page. *Nothing interesting to see yet.*

This is the current state of the repository at this moment [589688430e2dcd53e9ebf2a4aaa6ee008947b1b7](https://github.com/laravelista/laravel-api-101-consume/commit/589688430e2dcd53e9ebf2a4aaa6ee008947b1b7)²³¹.

The service class

The service class, that we will build has to do two things:

- Retrieve all articles
- Retrieve a single article

At the end of this chapter I will create methods on the service class to create, update and delete articles.

It is quite simple. We just need [Guzzle](https://github.com/guzzle/guzzle)²³². Run this command to install Guzzle:

```
composer require guzzlehttp/guzzle
```

In app, create a new folder called Api/Services and inside it create a new file Articles.php. Paste the following code inside:

²³¹<https://github.com/laravelista/laravel-api-101-consume/commit/589688430e2dcd53e9ebf2a4aaa6ee008947b1b7>

²³²<https://github.com/guzzle/guzzle>


```
<?php

namespace App\Services\Api;

use GuzzleHttp\Client as Guzzle;

class Articles
{
    protected $guzzle;

    protected $endpoint;

    public function __construct(Guzzle $guzzle)
    {
        $this->guzzle = $guzzle;
        $this->endpoint = config('services.api.endpoint') . '/articles';
    }
}
```

We have created a class called `Articles` which leverages dependency injection. In the constructor, we inject `Guzzle` and set the endpoint from the config file. The next step is to add the requested config key/value to the `config/services.php` file:

```
'api' => [
    'endpoint' => env('API_ENDPOINT')
]
```

Now in our `.env` file we have to set the key `API_ENDPOINT` to `http://localhost:8000/api/v1`. You could have hard-coded this value right into the class `Articles` or in the `config/services.php` file, but this way you can change the endpoint depending on the environment in which you are currently working (Useful for testing and local development).

```
API_ENDPOINT=http://localhost:8000/api/v1
```

Get All Articles

To retrieve all articles we have to send a GET request to `http://localhost:8000/api/v1/articles`, use `json_decode` to get an object from the response and finally use `collect()` to create a collection, on the body of the response object, that we can later manipulate at will.

Let's add a method to the `Articles` class called `all`:

```

public function all()
{
    $response = $this->guzzle->request('GET', $this->endpoint);

    $body = json_decode($response->getBody());

    return collect($body->data);
}

```

Remember, on the returned collection you can do everything as stated in the [documentation for Collections](#)²³³.

Get Single Article

This is very similar to retrieving all articles, but we will skip using `collect()` since we are only retrieving one object. Our current implementation of the API enables us to fetch articles by id. So we need to send a GET request to `http://localhost:8000/api/v1/articles/{id}`. Replacing the `{id}` with the id of the actual article that we want to retrieve.

For better SEO (Search Engine Optimization) it would be better that we use a `slug` field, instead of the `id`. Also using a direct id from the database could be considered a security risk.

Finally, we will use `json_decode` on the body of the response to get an object that we can work with.

Let's add a method to the `Articles` class called `get`:

```

public function get($id)
{
    $response = $this->guzzle->request('GET', $this->endpoint . "{/$id}");

    $body = json_decode($response->getBody());

    return $body->data;
}

```

Quick tip! If this method ran and it returned an error, you would get an error in your application. To avoid this from happening, you can create another method called `getOrFai1`, and inside it handle the situation if an error occurs during the request. See the [Guzzle documentation for Exceptions](#)²³⁴ to learn more.

²³³<https://laravel.com/docs/5.3/collections>

²³⁴<http://docs.guzzlephp.org/en/latest/quickstart.html#exceptions>

This is the current state of the repository at this moment [1fd0d3c41a073f04e081e56ed89658c942bf4ee3](https://github.com/laravelista/laravel-api-101-consume/commit/1fd0d3c41a073f04e081e56ed89658c942bf4ee3)²³⁵.

I will now quickly add methods for our API on the Article service class to create, update and delete articles. We won't be needing this for our tutorial, but it may be beneficial for you to see how this is done.

Create a new article

Before we continue with the actual code for creating a new article, there are some things that we need to setup. If you remember, endpoints for creating, updating and deleting an article are protected using a token based authentication. To be able to perform those actions we need to authenticate with the API. That means that we will need to pass a header to our request containing the token.

We will store the token in the same way that we stored the API endpoint. In the config/services.php config file which pulls the data from our .env file. Go to config/services.php and modify the api section to look like this:

```
'api' => [
    'endpoint' => env('API_ENDPOINT'),
    'token' => env('API_TOKEN')
]
```

Now in our .env file we have to set the key API_TOKEN to <your-token-here>.

```
API_TOKEN=cafbef5071387b4e4646732334dddff03
```

Remember to replace this value with your own token.

Let's add a method to the Articles class called create:

```
public function create(array $data)
{
    $this->guzzle->request('POST', $this->endpoint, [
        'form_params' => $data,
        'headers' => [
            'Authorization' => 'Bearer ' . config('services.api.token'),
        ],
    ]);

    return true;
}
```

²³⁵ <https://github.com/laravelista/laravel-api-101-consume/commit/1fd0d3c41a073f04e081e56ed89658c942bf4ee3>

This method accepts an array of data. *Remember which fields are required?*

If the article is created or the response code is anything in the 200 range, it will return true.

Example

From your ArticleController you could do something like:

```
$this->articles->create([
    'heading' => 'test',
    'content' => 'test',
    'category_id' => 10
]);
```

Remember that creating, updating and deleting of articles should be done by a authenticated and authorized user of that site.

Update an existing article

Updating an article is very similar to creating one. You will just have to pass the id of the article that you want to update and the data.

Let's add a method to the Articles class called update:

```
public function update($id, array $data)
{
    $this->guzzle->request('PUT', $this->endpoint .("/{ $id }", [
        'form_params' => $data,
        'headers' => [
            'Authorization' => 'Bearer ' . config('services.api.token'),
        ],
    ]);

    return true;
}
```

Example

From your ArticleController you could do something like:

```
$article_id = 7;

$this->articles->update($article_id, [
    'heading' => 'test',
    'content' => 'test'
]);
```

There are no required fields for updating an article on the API.

Delete an article

Deleting an article is also very simple.

Let's add a method to the Articles class called delete:

```
public function delete($id)
{
    $this->guzzle->request('DELETE', $this->endpoint . "/{$id}", [
        'headers' => [
            'Authorization' => 'Bearer ' . config('services.api.token'),
        ],
    ]);

    return true;
}
```

This time, we only need the id of the article and it is deleted.

Example

From your ArticleController you could do something like:

```
$this->articles->delete(7);
```

This is the current state of the repository at this moment [22a023cbc7d9fb0ae9f29faf951d9fa6f4615ab0](https://github.com/laravelista/laravel-api-101-consume/commit/22a023cbc7d9fb0ae9f29faf951d9fa6f4615ab0)²³⁶.

Integration

Now that we have our Article Service class, we have to integrate it in our application.

Tip If this Article Service class is going to be used in other Laravel application as well, it would be better to create a package from it. Then, you can simply install it using Composer and avoid duplicating the code in each application. Also, it is much easier to add new functionality to it.

²³⁶ <https://github.com/laravelista/laravel-api-101-consume/commit/22a023cbc7d9fb0ae9f29faf951d9fa6f4615ab0>

Article Controller

We will use dependency injection (DI) here to inject our Articles service class and make it available to all the methods inside the controller. Add this constructor to app/Http/Controllers/Article-Controller class.

```
protected $articles;

public function __construct(Articles $articles)
{
    $this->articles = $articles;
}
```

Remember to add a use statement above the class: use App\Services\Api\Articles;.

Now, modify the methods index and show, so that they look like this:

```
public function index()
{
    $articles = $this->articles->all();

    return view('articles.index')->with(compact('articles'));
}

public function show($id)
{
    $article = $this->articles->get($id);

    return view('articles.show')->with(compact('article'));
}
```

In the index method we are fetching all articles and passing them as a variable to our view articles.index. In the show method we are fetching a single article by its id and passing it to our view articles.show. *Piece of cake!*

Views

Now that our views are receiving articles/article, we have to update them to display needed information.

Home Page

For our home view which is located in resources/views/articles/index.blade.php, just below the h1 tag, add this code:

```

@forelse($articles as $article)
    <div class="panel panel-default">
        <div class="panel-body">
            <h2><a href="{ route('articles.show', $article->id) }}">{{ $article\
->heading }}</a></h2>
            <p>{{ str_limit($article->content, 150) }}</p>
        </div>
    </div>
@empty
    <p class="lead">No articles yet!</p>
@endforelse

```

This code loops thru all the given articles, displays their heading and first 150 characters of the content. Provided in the heading is the link to view the whole article. This uses a route name which we declared at the start of this tutorial. If there are no articles, it will display a message saying so.

You can modify this however you want.

Article Page

For our article view which is located in `resources/views/articles/show.blade.php`, replace the line where `h1` tag is located with:

```

<h1>{{ $article->heading }}</h1>
<p>{{ $article->content }}</p>

```

Also, it would be nice to display the article heading in the tab of the page, so change the section title to `$article->heading`:

```
@section('title', $article->heading)
```

This is the current state of the repository at this moment [46cd4beaf758250a13ec5a09c8cdc527e7dab0f4](https://github.com/laravelista/laravel-api-101-consume/commit/46cd4beaf758250a13ec5a09c8cdc527e7dab0f4)²³⁷.

This completes this tutorial. You now have a working Article service class and a basic demonstration on how to include the data from our API in a Laravel application. In the next tutorial, we will add tests to our API.

²³⁷ <https://github.com/laravelista/laravel-api-101-consume/commit/46cd4beaf758250a13ec5a09c8cdc527e7dab0f4>

TEST

Laravel comes out-of-the-box with great testing suite built on PHPUnit. No API should be left untested.

Published at: 18. December, 2016.



View Source Code

Source code for this tutorial is available [here](#)²³⁸.

The hardest part about starting a new tutorial is writing an introduction.

In the first part of this serial called [BUILD](#), we have built an API for managing news articles. In this tutorial, we will write tests for that API using the PHPUnit testing framework that comes with Laravel by default.

First, we need to set up our testing environment, then we will write the tests following the documentation on [Testing JSON APIs](#)²³⁹.

We need to test five routes:

- index - get all articles
- store - create an article with authentication
- show - get a single article
- update - update an article with authentication
- delete - delete an article with authentication

Setting up the environment

This is the current state of the repository at this moment [66e36fc1517ba6bc7fa4706369ef8a8b8d60caab](#)²⁴⁰.

All of the tests will be located in the tests directory. Now, if you take a look there, you will see two files: `ExampleTest.php` and `TestCase.php`. Delete the `ExampleTest.php` file. It is just an example test that comes with Laravel out-of-the-box. The `TestCase.php` is the important one. It bootstraps our application and provides us with a convenient helper methods, that allow us to expressively test our applications.

Environment variables

Testing environment variables may be configured in the `phpunit.xml` file. By default, these are the values with which it comes with:

²³⁸<https://github.com/laravelista/laravel-api-101-build>

²³⁹<https://laravel.com/docs/5.2/testing#testing-json-apis>

²⁴⁰<https://github.com/laravelista/laravel-api-101-build/commit/66e36fc1517ba6bc7fa4706369ef8a8b8d60caab>


```
<env name="APP_ENV" value="testing"/>
<env name="CACHE_DRIVER" value="array"/>
<env name="SESSION_DRIVER" value="array"/>
<env name="QUEUE_DRIVER" value="sync"/>
```

As you can see, the configuration environment is set to testing and cache, session and queue are set to use array drivers.

:memory database

Our API stores data into an SQLite database, so we will use an in-memory SQLite database for testing. It will be even quicker than using an SQLite file database. To do so, go to `config/database.php` and under connections add a new connection like so:

```
'testing' => [
    'driver'   => 'sqlite',
    'database' => ':memory:',
    'prefix'   => '',
],
```

Now, we need to tell PHPUnit that we want to use that connection for testing. Open `phpunit.xml` file and add a new environment variable:

```
<env name="DB_CONNECTION" value="testing"/>
```

This way, any time we run `vendor/bin/phpunit` the tests will use our testing connection for the database.

We can do a dry run now. Run `vendor/bin/phpunit` from the command line:

```
$ vendor/bin/phpunit
PHPUnit 4.8.26 by Sebastian Bergmann and contributors.
```

```
Time: 2 seconds, Memory: 6.00MB
```

```
No tests executed!
```

This is the current state of the repository at this moment [f583205a9f50eab91cdf6174d0021f434557f00b](https://github.com/laravelista/laravel-api-101-build/commit/f583205a9f50eab91cdf6174d0021f434557f00b)²⁴¹.

²⁴¹<https://github.com/laravelista/laravel-api-101-build/commit/f583205a9f50eab91cdf6174d0021f434557f00b>

Writing tests

We will only need one file for all our tests. It is wise to group tests by resources if possible. We will test the articles part of the API since that is the only part that we have built.

Defining tests

To create a new test file, run this command from the command line:

```
php artisan make:test ArticleTest
```

A new file will be created in tests directory called ArticleTest.php. Inside it, you will find a testExample method. We won't be needing that, so go ahead and delete that method from the file.

Database migrations

One thing that we will be needing is the DatabaseMigrations trait, so go ahead and add that to our ArticleTest class.

```
<?php
```

```
use Illuminate\Foundation\Testing\WithoutMiddleware;  
use Illuminate\Foundation\Testing\DatabaseMigrations;  
use Illuminate\Foundation\Testing\DatabaseTransactions;
```

```
class ArticleTest extends TestCase  
{  
    use DatabaseMigrations;  
}
```

This trait is used to rollback the database after each test and migrate it before the next test. Very handy.

Get all articles test

We will now add a new method to ArticleTest. Copy the code bellow and be sure to include the comment:

```
/**
 * @test
 */
public function it_gets_all_articles()
{
    // code goes here
}
```

If you omit the comment, specifically @test part, PHPUnit will not detect the test. It presumes that all test methods are prefixed with test. By adding @test in the comment, we are instructing PHPUnit to treat this method as a test method.

For this test to work, we will populate the database with 10 articles, using the factory helper method. This method is a part of [Model Factories](#)²⁴². Model factories are a great way to specify the value of each column for a model. If you take a look at database/factories/ModelFactory.php, you will see that I have already defined factories for each model that we have in our API. If you are writing the code as we go along, be sure to copy the content of [ModelFactory from Github](#)²⁴³.

First, we will populate the database. Replace the comment with this code:

```
factory(\App\Article::class, 10)->create([
    'published' => true
]);
```

Then, we need to verify that the status code is 200 and that the returned JSON structure is what we expect. Below the factory function place this code:

```
$this->json('GET', '/api/v1/articles')
->assertResponseStatus(\Illuminate\Http\Response::HTTP_OK) // 200
->seeJsonStructure([
    'data' => [
        '*' => [
            'id',
            'heading',
            'content',
            'published',
            'published_at',
            'tags'
        ]
    ]
]);
```

Now, run vendor/bin/phpunit from the command line:

²⁴²<https://laravel.com/docs/5.2/testing#model-factories>

²⁴³<https://github.com/laravelista/laravel-api-101-build/blob/3dd4d8c828d9e64724214588d548854999fd14ce/database/factories/ModelFactory.php>

```
$ vendor/bin/phpunit
PHPUnit 4.8.26 by Sebastian Bergmann and contributors.
```

```
.
```

```
Time: 2.36 seconds, Memory: 18.00MB
```

```
OK (1 test, 63 assertions)
```

For some reason, here is where I got this error ‘ Class ‘DoctrineDBALDriverPDOMySqlDriver’ not found. Cannot really tell why, but I have found a solution. Do composer require doctrine/dbal’ from the command line if this happens to you too.

Great! We have our first test and it returns green. We still have to write another test for ?include=category.Member?

Bellow it_gets_all_articles method, add another method and call it it_gets_all_articles_with_category:

```
/**
 * @test
 */
public function it_gets_all_articles_with_category()
{
    factory(\App\Article::class, 10)->create([
        'published' => true
    ]);

    $this->json('GET', '/api/v1/articles?include=category')
        ->assertResponseStatus(\Illuminate\Http\Response::HTTP_OK) // 200
        ->seeJsonStructure([
            'data' => [
                '*' => [
                    'id',
                    'heading',
                    'content',
                    'published',
                    'published_at',
                    'tags',
                    'category' => [
                        'data' => [
                            'name'
                        ]
                    ]
                ]
            ]
        ]);
}
```

```

    ]
    ]
    ]
    ]);
}

```

Run `vendor/bin/phpunit` from the command line, and you get:

```

$ vendor/bin/phpunit
PHPUnit 4.8.26 by Sebastian Bergmann and contributors.

```

```

..

```

```

Time: 2.66 seconds, Memory: 18.00MB

```

```

OK (2 tests, 156 assertions)

```

This is the current state of the repository at this moment [87db84583b832cb9cd21cb2b44545a5463485d8b](https://github.com/laravelista/laravel-api-101-build/commit/87db84583b832cb9cd21cb2b44545a5463485d8b)²⁴⁴.

Get a single article test

This test is very similar to get all articles test, but here we are interested in only one article. That is why we will populate the database with only one record.

Create a new test method called `it_gets_single_article` in `tests/ArticleTest.php`:

```

/**
 * @test
 */
public function it_gets_single_article()
{
    // Code goes here
}

```

Place the following code inside:

²⁴⁴<https://github.com/laravelista/laravel-api-101-build/commit/87db84583b832cb9cd21cb2b44545a5463485d8b>

```
$article = factory(\App\Article::class)->create([
    'published' => true
]);
```

Here, we create only one article and assign it to a variable for later use.

Now below that, paste the following:

```
$this->json('GET', "/api/v1/articles/{$article->id}")
->assertResponseStatus(\Illuminate\Http\Response::HTTP_OK) // 200
->seeJsonStructure([
    'data' => [
        'id',
        'heading',
        'content',
        'published',
        'published_at',
        'tags'
    ]
]);
```

Here we make a GET request to /api/v1/articles/<article-id>, verify that the response code is 200 and that the returns JSON structure is as we expect it to be.

Member the ?include=category? I member.

Let's add another method called it_gets_single_article_with_category:

```
/**
 * @test
 */
public function it_gets_single_article_with_category()
{
    $article = factory(\App\Article::class)->create([
        'published' => true
    ]);

    $this->json('GET', "/api/v1/articles/{$article->id}?include=category")
        ->assertResponseStatus(\Illuminate\Http\Response::HTTP_OK) // 200
        ->seeJsonStructure([
            'data' => [
                'id',
                'heading',
```

```

        'content',
        'published',
        'published_at',
        'tags',
        'category' => [
            'data' => [
                'name'
            ]
        ]
    ]
});
}

```

Run `vendor/bin/phpunit` from the command line, and you get:

```

$ vendor/bin/phpunit
PHPUnit 4.8.26 by Sebastian Bergmann and contributors.

....

Time: 3.3 seconds, Memory: 18.00MB

OK (4 tests, 175 assertions)

```

This is the current state of the repository at this moment [d8a9472145dae101c17c4b731b29baf4b0f06e8b](https://github.com/laravelista/laravel-api-101-build/commit/d8a9472145dae101c17c4b731b29baf4b0f06e8b)²⁴⁵.

Create article test

To write a test that will test creating a new article, we need to create a user, obtain the `api_token`, send the token with the request, post the required data, verify the response code 201 and verify the JSON response.

Let's start by adding a new test method called `it_creates_a_new_article` in `tests/ArticleTest.php`:

²⁴⁵<https://github.com/laravelista/laravel-api-101-build/commit/d8a9472145dae101c17c4b731b29baf4b0f06e8b>

```
/**
 * @test
 */
public function it_creates_a_new_article()
{
    // Code goes here...
}
```

Inside it, we first need to create a new user and a category:

```
$user = factory(\App\User::class)->create();

$category = factory(\App\Category::class)->create();
```

Then, we will prepare the data that we will post with the request:

```
$data = [
    'heading' => 'Test heading',
    'content' => 'Test content',
    'category_id' => $category->id,
    // Optional
    'published' => "1",
    'published_at' => \Carbon\Carbon::now()->toDateTimeString(),
];
```

Finally, we make a POST request to `/api/v1/articles` with the `api_token` as query parameter, and the data that we have prepared:

```
$this->post("/api/v1/articles?api_token={$user->api_token}", $data)
    ->assertResponseStatus(\Illuminate\Http\Response::HTTP_CREATED) // 201
    ->seeJson([
        'message' => 'Created'
    ])
    ->seeInDatabase('articles', $data);
```

We verify the response code, look for "message": "Created" in JSON response and check in the database for an article with given data.

Run `vendor/bin/phpunit` from the command line, and you get:


```
$ vendor/bin/phpunit
PHPUnit 4.8.26 by Sebastian Bergmann and contributors.
```

```
.....
```

```
Time: 3 seconds, Memory: 20.00MB
```

```
OK (5 tests, 179 assertions)
```

This is the current state of the repository at this moment [5f188abd23f43d52e517a52a4705d320c76559f4](https://github.com/laravelista/laravel-api-101-build/commit/5f188abd23f43d52e517a52a4705d320c76559f4)²⁴⁶.

Update article test

Since all fields are optional, we can test this with only one field. First, we will need to create an article in the database. Then, we will send a PUT request with the article id and the api_token, and pass a new heading for the article. We expect to receive status code 203, see JSON contains "message": "Updated" and verify in the database that the article has had its heading changed.

Add a new test method called `it_updates_an_existing_article` in `tests/ArticleTest.php`:

```
/**
 * @test
 */
public function it_updates_an_existing_article()
{
    // Code goes here...
}
```

First, we create a user and an article:

```
$user = factory(\App\User::class)->create();

$article = factory(\App\Article::class)->create([
    'heading' => 'Batman',
    'published' => true
]);
```

Then, we do the rest:

²⁴⁶<https://github.com/laravelista/laravel-api-101-build/commit/5f188abd23f43d52e517a52a4705d320c76559f4>

```

$this->put("/api/v1/articles/{ $article->id }?api_token={ $user->api_token }", [
    'heading' => 'Superman'
])
->assertResponseStatus(\Illuminate\Http\Response::HTTP_ACCEPTED) // 202
->seeJson([
    'message' => 'Updated'
])
->seeInDatabase('articles', [
    'id' => $article->id,
    'heading' => 'Superman'
]);

```

You can verify that this test returns green by running `vendor/bin/phpunit` from the command line:

```

$ vendor/bin/phpunit
PHPUnit 4.8.26 by Sebastian Bergmann and contributors.

```

```

.....

```

```

Time: 3.19 seconds, Memory: 20.00MB

```

```

OK (6 tests, 183 assertions)

```

This is the current state of the repository at this moment [dda4d138cd36a5e77256a2a1f6292969aa02c06e](https://github.com/laravelista/laravel-api-101-build/commit/dda4d138cd36a5e77256a2a1f6292969aa02c06e)²⁴⁷.

Delete article test

The only thing left now, is to test deleting an article. Let's make this one quick. Paste the following code inside the `ArticleTest` class:

```

/**
 * @test
 */
public function it_deletes_an_article()
{
    $user = factory(\App\User::class)->create();

    $article = factory(\App\Article::class)->create();

    $this->delete("/api/v1/articles/{ $article->id }?api_token={ $user->api_token }")

```

²⁴⁷ <https://github.com/laravelista/laravel-api-101-build/commit/dda4d138cd36a5e77256a2a1f6292969aa02c06e>

```

->assertResponseStatus(\Illuminate\Http\Response::HTTP_OK) // 200
->seeJson([
    'message' => 'OK'
])
->dontSeeInDatabase('articles', [
    'id' => $article->id
]);
}

```

We create a user and an article. Then we send a DELETE request with the article id and users' api-token. We expect to see status code 200, JSON message saying OK, and finally we expect not to see in the database an article with the given id.

Run all tests now with vendor/bin/phpunit, to see that everything passes:

```
$ vendor/bin/phpunit
PHPUnit 4.8.26 by Sebastian Bergmann and contributors.
```

```
.....
```

```
Time: 3.44 seconds, Memory: 20.00MB
```

```
OK (7 tests, 187 assertions)
```

This is the current state of the repository at this moment [52f4dc120d813c98e44bfb3ae8943bc0253dd89a](https://github.com/laravelista/laravel-api-101-build/commit/52f4dc120d813c98e44bfb3ae8943bc0253dd89a)²⁴⁸.

This completes this tutorial. Our API built in the first tutorial is now tested. You can now use this knowledge to test your future APIs' built with Laravel. In the next tutorial, I will show you how to write awesome documentation for our API.

DOCUMENT

With API Blueprint you can quickly design and prototype APIs to be created or document and test already deployed mission-critical APIs.

Published at: 19. March, 2017.



View Source Code

Source code for this tutorial is available [here](https://github.com/laravelista/laravel-api-101-build/commit/52f4dc120d813c98e44bfb3ae8943bc0253dd89a)²⁴⁹.

²⁴⁸<https://github.com/laravelista/laravel-api-101-build/commit/52f4dc120d813c98e44bfb3ae8943bc0253dd89a>

²⁴⁹<https://gist.github.com/mabasic/ce4fa4f7608daded251aea6ac6989a8c>

It is about time that we finish this course on building an API with Laravel. We have built an API, added authorization, written an application that uses our API, and written tests for our API. Now, in this last step, **we will write documentation for it.**

Documenting your code, application, product, API or whatever is a very important task, no matter if you are working alone or in a group. **As long as it is something worth documenting, you should write documentation for it.** It happened to me so many times. I would create something and in a couple of weeks, I would forget everything about how it works and what it does. Then I would have to read the source code to figure it out. *Don't do this.*

It is my opinion that it does not matter if you write documentation first, during coding or after everything has finished, as long as you provide documentation at some point.

To document our API we will use [API Blueprint²⁵⁰](#) language. The reason why it took me so long to complete this course was because I wanted to provide a course on API Blueprint before completing this course.

Prerequisite! To learn what is API Blueprint and how to use it see my course [Write better API documentation with API Blueprint](#).

Document this!

In this tutorial, we are documenting an API, and in an API it is all about endpoints and authorization. We need to document every available endpoint and how to use endpoints that require authorization.

We will write a chapter on authorization and different ways of authorizing:

- query parameter
- request body
- header authorization bearer

Then, we will cover the endpoints:

- index
- create
- show
- store
- update
- destroy

The purpose of documenting is not to write as much text as possible, but to write as little as it is needed to explain.

²⁵⁰<https://apiblueprint.org/>

API Blueprint

If you haven't already, at this point you really should read [Write better API documentation with API Blueprint](#).

Let's create a file and call it `articles.apib`.

Inside this file, let's add the required API Blueprint version, our API title, and a short description:

```
FORMAT: 1A
```

```
# Articles
```

```
Simple API for managing news articles.
```

Great!

Authorization

Now to add a chapter about authorization. I will reuse the text for authorization, from the [AUTH](#) tutorial in this course and modify it a bit.

```
# Authorization
```

```
To create, update or destroy an article, you have to authorize. To authorize you\
  have to send your personal `api_token` key with the request in one of these thr\
ee possible ways:
```

```
## Query parameter
```

```
Append `?api_token=<your_token_here>` to the end of the URL `POST /api/v1/articles`.
```

```
## Request body
```

```
To the body of the request, along with other data like `heading` and `content`, \
pass `api_token` with the value of your token. Be sure to use `x-www-form-urlencoded` in Postman.
```

```
## Header Authorization Bearer
```

```
Add a header to the request for `Authorization:Bearer <your_token_here>`.
```

In Postman click on Headers tab and start typing in key field `Authorization`, a drop-down will show so that you can select it. In value field type `Bearer <your_token_here>`. Be sure to replace ``<your_token_here>`` with the actual token you got from seeding the user.

Endpoints

Since we only have endpoints related to articles, we will group those endpoints in a group by writing:

```
# Group Articles
```

In that group, we will have two resources: *Article Collection* and *Article*. *Article Collection* resource will contain actions on how to view all articles and create a new article. *Article* resource will contain actions on how to view a single article, update an existing article and destroy an article.

The reason behind why we have two resources and not a single resource, is because API Blueprint handles resources in a specific way. Each resource is tied to a specific URI. So, for viewing all articles and creating a new article it is `/api/v1/articles`, and for viewing a single article, updating and deleting an article it would be `/api/v1/articles/{article_id}`. Only the request method and body is different.

Article Collection

Let's create a resource called *Article Collection*:

```
## Article Collection [/api/v1/articles{?include}]
```

This resource allows you to view a list of articles and create a new article.

View all articles

Now to add an action for viewing all articles, add this text:

```
### View all articles [GET]
```

This action returns all articles in JSON format with tags.

By default, it does not return the article category.

You have to ask for it specifically by using `&include=category`.

```
+ Parameters
```

```
+ include (optional, string) - Items that should be included in the result. \
```

Can be: category.

+ Response 200 (application/json)

```
{
  "data": [
    {
      "id": 1,
      "heading": "Et magni quam necessitatibus.",
      "content": "Tenetur qui corporis blanditiis non ipsam aliqui\nd. Sunt autem qui unde modi molestias. Nam vel rerum minima similique impedit vo\luptates voluptatum sit.",
      "published": true,
      "published_at": "12.03.1977",
      "tags": {
        "data": [
          {
            "name": "dolores"
          },
          {
            "name": "nihil"
          },
          {
            "name": "maxime"
          }
        ]
      }
    },
    {
      "id": 2,
      "heading": "Blanditiis ea dignissimos rerum at enim.",
      "content": "Est numquam quos architecto minus. Aut voluptate\m porro explicabo quidem ipsum quas. Dignissimos non voluptatem atque. Aut ea ne\que aperiam non numquam vero dolores.",
      "published": true,
      "published_at": "12.11.1998",
      "tags": {
        "data": []
      }
    }
  ]
}
```

Create a new article

To add an action for creating a new article, add this text:

```
### Create a new article [POST]
```

You may create a new article using this action. **This action requires authentication.**

```
+ heading (string, required) - Article title.
+ content (text, required) - Article content.
+ category_id (number, required) - ID of the category to which the article belongs.
+ published_at (date) - DateTime when the article is published `2017-03-19 20:47:00`.
+ published (boolean) - Indicates if the article is published `1` or `0`.
```

```
+ Request (application/json)
```

```
  + Headers
```

```
    Authorization: Bearer your-token-here
```

```
  + Body
```

```
    {
      "heading": "How to win the lottery",
      "content": "Wouldn't you like to know ;)",
    }
```

```
+ Response 201 (application/json)
```

```
  {
    "message": "Created",
    "status_code": 201
  }
```

We are now finished with *Article Collection* resource.

Article

Let's create a resource called *Article*:


```
## Article [/api/v1/articles/{article_id}{?include}]
```

This resource allows you to view a single article, update an article or destroy \n an article.

+ Parameters

+ article_id (string) - The ID of the article on which you want to do action \n s on.

View a single article

To add an action for viewing a single article, add this text:

```
### View a single article [GET]
```

If you want to fetch data for a single article this is what you will want to use.

+ Parameters

+ include (optional, string) - Items that should be included in the result. \n Can be: category.

+ Response 200 (application/json)

```
{
  "data": [
    {
      "id": 1,
      "heading": "Et magni quam necessitatibus.",
      "content": "Tenetur qui corporis blanditiis non ipsam aliqui\nd. Sunt autem qui unde modi molestias. Nam vel rerum minima similique impedit vo\luptates voluptatum sit.",
      "published": true,
      "published_at": "12.03.1977",
      "tags": {
        "data": [
          {
            "name": "dolores"
          },
          {
            "name": "nihil"
          },
          {
            "name": "maxime"
          }
        ]
      }
    }
  ]
}
```

```

    }
  ]
}

```

Update an existing article

To add an action for updating an existing article, add this text:

```
### Update an existing article [PUT]
```

This action enables you to update an existing article. ****This action requires authentication.****

You can send all parameters in the request body or just the parameters that you want to change.

```

+ heading (string) - Article title.
+ content (text) - Article content.
+ category_id (number, required) - ID of the category to which the article belongs.
+ published_at (date) - DateTime when the article is published `2017-03-19 20:47\
:00`.
+ published (boolean) - Indicates if the article is published `1` or `0`.

```

```
+ Request (application/json)
```

```
+ Headers
```

```
Authorization: Bearer your-token-here
```

```
+ Body
```

```

{
  "category_id": "2",
}

```

```
+ Response 202 (application/json)
```

```

{
  "message": "Updated",
}

```

```

        "status_code": 202
    }

```

Delete an existing article

To add an action for deleting an existing article, add this text:

```
### Delete an existing article [DELETE]
```

This action enables you to delete an existing article. ****This action requires authentication.****

+ Request (application/json)

+ Headers

```
Authorization: Bearer your-token-here
```

+ Response 202 (application/json)

```

{
    "message": "OK",
    "status_code": 200
}

```

And finish!

That is it! We now have documentation for our API. You can view the entire source code for the blueprint [here](#)²⁵¹.

Also, if you are using a service like Apiary, you will get pretty documentation, like this one [here](#)²⁵².

This tutorial concludes this entire course on Laravel API 101.

²⁵¹<https://gist.github.com/mabasic/ce4fa4f7608daded251aea6ac6989a8c>

²⁵²<http://docs.articles13.apiary.io/#>

Write better API documentation with API Blueprint

API Blueprint is a description language for web APIs. Quickly design and create API prototypes, or document and test already deployed APIs.

API Blueprint introduction

Write better API documentation. Learn what is API Blueprint and what are its use cases.

Published at: 26. February, 2017.



View Source Code

Source code for this tutorial is available [here](https://raw.githubusercontent.com/apiaryio/api-blueprint/master/examples/Polls%20API.md)²⁵³.

Let's say that you are given a task of documenting an API for other developers. What would you do?

This is what I have done in the past, and this is what I think most people in the same situation would do.

I created a markdown document, called it `api.md` and gave it a title `API documentation`. Inside it, I added a new chapter for each resource, and for each resource, there were subchapters for each action that was available for that resource. Then, for each action, I wrote down the URI, required and optional attributes, a sample response, and a short description of what it does.

Sounds familiar?

If you are currently doing that or have done that in the past, congrats! Almost any documentation is better than none. There is nothing wrong in doing it in that way, especially if you are a solo developer.

If you are working in a team where anyone can contribute to the documentation, this is where things start to break down. The main issue is inconsistency. Since you are using regular markdown to describe the API, other developers will probably use their own style to write the same things, and you will end up with documentation that is very hard to read, understand and navigate. Also, over time you may forget how you structured your own document, and mess it up yourself.

Don't worry, there is a solution.

²⁵³<https://raw.githubusercontent.com/apiaryio/api-blueprint/master/examples/Polls%20API.md>

API Blueprint

In this tutorial I will introduce you to [API Blueprint](#)²⁵⁴. API Blueprint is a powerful high-level API description language for web APIs.

“API Blueprint is simple and accessible to everybody involved in the API lifecycle. Its syntax is concise yet expressive. With API Blueprint you can quickly design and prototype APIs to be created or document and test already deployed mission-critical APIs.” - [source](#)²⁵⁵

Its main features are:

- focused on collaboration
- open source (MIT license)
- recognized by GitHub (file extension .apib)
- better API designs through abstraction
- design-first philosophy
- awesome tools

If you want to get started with API Blueprint right away you can check out the following links. They have everything you need to know about the language and how to use it:

- [API Blueprint Tutorial](#)²⁵⁶
- [API Blueprint Advanced Tutorial](#)²⁵⁷
- [API Blueprint Specification](#)²⁵⁸
- [API Blueprint Examples](#)²⁵⁹

Awesome tools

This is the most interesting feature of the API Blueprint. Once you write down the blueprint (document), you have tools which understand the blueprint, and according to it they can build your API, test an existing API, provide you with a mocked version of the API that you wrote down or generate HTML documentation.

You can view on the [Tools page](#)²⁶⁰ what tools are available. I will just mention few that I think are worth looking into:

²⁵⁴<https://apiblueprint.org/>

²⁵⁵<https://apiblueprint.org/>

²⁵⁶<https://apiblueprint.org/documentation/tutorial.html>

²⁵⁷<https://apiblueprint.org/documentation/advanced-tutorial.html>

²⁵⁸<https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md>

²⁵⁹<https://github.com/apiaryio/api-blueprint/tree/master/examples>

²⁶⁰<https://apiblueprint.org/tools.html>

- [Apiary](#)²⁶¹
- [API Blueprint Sublime Text plugin](#)²⁶²
- [API Blueprint Grammar for Atom](#)²⁶³
- [API Blueprint Preview](#)²⁶⁴
- [Atom Linter for API Blueprint](#)²⁶⁵
- [Dredd](#)²⁶⁶
- [Snowboard](#)²⁶⁷
- [Agllo](#)²⁶⁸
- [Sandbox](#)²⁶⁹
- [Api-Mock](#)²⁷⁰

Use cases

Here are a couple of situations where I have used API Blueprint in the past:

Quick prototype

I had to create a quick prototype of the API which would be consumed by a mobile application.

In the beginning, we did not know what data we would need, so I used [Apiary](#)²⁷¹ to write a simple API and get the mocked version of it available for testing. The mobile app developer used the mocked version of the API to build the application; we had to do a few changes to the API as the application progressed.

When we were both happy with how it all works, I created the API using Laravel. He just had to change the endpoint of the API on his end and the mobile application was working.

Consistent implementation

There was an outdated Laravel application which needed to be completely rewritten.

I decided that it was best that I separate the frontend from the backend completely and create a SPA (single page application). I used [React.js](#)²⁷² for the frontend.

²⁶¹<https://apiary.io/>

²⁶²<https://github.com/apiaryio/api-blueprint-sublime-plugin>

²⁶³<https://atom.io/packages/language-api-blueprint>

²⁶⁴<https://atom.io/packages/api-blueprint-preview>

²⁶⁵<https://github.com/zdne/linter-api-blueprint>

²⁶⁶<https://github.com/apiaryio/dredd>

²⁶⁷<https://github.com/subosito/snowboard>

²⁶⁸<https://github.com/danielgtaylor/aglio>

²⁶⁹<https://getsandbox.com/>

²⁷⁰<https://github.com/localmed/api-mock>

²⁷¹<https://apiary.io/>

²⁷²<https://facebook.github.io/react/>

The reason why I separated the frontend completely from the backend was because that enabled me to rewrite the backend behind the scenes, while the new and shiny frontend was available to the client.

I created the documentation for the API which was to be created. During the application rewriting, I simply implemented the API according to the documentation, and the switch from the old to the new application was seamless.

Thank you for reading. In the next tutorial, we will go over API Blueprint basics.

API Blueprint basics

This tutorial will take you through the basics of the API Blueprint language.

Published at: 05. March, 2017.



View Source Code

Source code for this tutorial is available [here](#)²⁷³.

Well hello there! Welcome to the second tutorial in the “Write better API documentation with API Blueprint” course. In this tutorial, we will build an API blueprint step by step for an imaginary API called “Bookstore”. Bookstore is a simple API which allows consumers to view all books and create new ones.

Before we begin there is one thing that you have to remember. API Blueprint can do a lot more than is going to be shown here. I will show you the basics now, and in the next chapter, we will improve upon this blueprint.

Let’s get started

API Blueprint is a plain text file with an extension `.apib`, but you can also use the `.md` (markdown) extension.

You can use a plain text editor like Notepad, or something with a more bang like Sublime Text or Atom. I personally use Atom, because it supports a few helpful packages for working with API Blueprints.

Atom API Blueprint packages:

- [API Blueprint Grammar for Atom](#)²⁷⁴ - Syntax highlighting for API Blueprint. The only one that works on Windows OS.

²⁷³<https://gist.github.com/mabasic/acd638280aa7b41eabf2d1c737b8cc83>

²⁷⁴<https://atom.io/packages/language-api-blueprint>

- [API Blueprint Preview](#)²⁷⁵ - Provides nice HTML preview like when editing markdown in Atom. Works only on Linux and Mac OS.
- [Atom Linter for API Blueprint](#)²⁷⁶ - Automatically lints your API Blueprint. Works only on Linux and Mac OS.

Sublime Text API Blueprint packages:

- [API Blueprint Sublime Text plugin](#)²⁷⁷ - Provides linter and syntax highlighting.

You don't need these packages or text editors to work with API Blueprints, but I highly suggest that you install one at this point. *API Blueprint Grammar for Atom* is a must have if you are using Atom text editor.

Basics

Let's create a new folder called Bookstore, and inside it create an empty file `bookstore.apib`. Everything we do from this point on is going to happen in that file.

Metadata

"Metadata keys and their values are tool-specific. Refer to relevant tool documentation for the list of supported keys." - [Metadata documentation](#)²⁷⁸

At the top of the document add `FORMAT: 1A`. This sets the version of the API Blueprint.

```
FORMAT: 1A
```

You can also add `HOST` key, which in certain tools like Apiary sets the URL on which the actions are to be performed.

Example:

```
FORMAT: 1A
HOST: http://polls.apibluprint.org/
```

Name & description

After the metadata goes the API name and description.

²⁷⁵<https://atom.io/packages/api-blueprint-preview>

²⁷⁶<https://github.com/zdne/linter-api-blueprint>

²⁷⁷<https://github.com/apiaryio/api-blueprint-sublime-plugin>

²⁷⁸<https://github.com/apiaryio/api-blueprint/blob/master/API%20Blueprint%20Specification.md#metadata-section>


```
# Bookstore
```

Bookstore is a simple API which allows consumers to view all books and create new ones.

Chapters

Before proceeding to group resources, resources and actions, I like to have a few chapters for all the other stuff. Like if you have authorization or if the caching is enabled, or if you provide multilingual content on the API and similar.

If you think that there is something that a potential reader of this blueprint or consumer should know, then this is the place to write it.

Example:

```
# Overview
```

This API is open to the public. Anyone can view all books, and anyone can create new books. There is no language filter so be aware. This API is for demonstration purposes only.

Resource groups

Every API has its resources. In API Blueprint you can group related resources. In our Bookstore API we have a Book resource, so we will create a Book resource group like so:

```
# Group Book
```

Resources related to books in the API.

You can have as many resource groups as you want. Inside *resource groups* go *resources*, and inside resources go *actions*.

Resources

We will have a resource for *Book collection*. This resource is used to view a list of all books and create a new book.

```
## Book collection [/books]
```

This resource allows you to view a list of books and create a new book.

Resource must be a sub-heading of the resource group or standalone.

In brackets at the end of the heading is specified a URI which is used to access the resource. Every action inside this resource will be conducted on that URI.

For this example, we don't need another resource, but you could have one for *Book* which would allow you to view a single book, update it or delete it.

Actions

Actions are written as sub-headings of the resource that they belong to followed by the HTTP method.

We will have two actions on this resource. The GET actions to *View all books*, and the POST action to *Create a new book*.

View all books:

```
### View all books [GET]
```

This action returns all books in JSON format.

+ Response 200 (application/json)

```
{
  "data": [
    {
      "title": "A Collection of Laravel Tutorials",
      "author": "Mario Bašić",
      "year_published": 2017,
      "completed": false
    }
  ]
}
```

Create a new book:

```
### Create a new book [POST]
```

You may create a new book using this action. It takes a JSON object containing a title and an author.

```
+ title (string) - The title of the book.
+ author (string) - Name of the author.
```

```
+ Request (application/json)
```

```
{
  "title": "The answer to everything",
  "author": "Mario Bašić",
}
```

```
+ Response 201 (application/json)
```

```
{
  "title": "The answer to everything",
  "author": "Mario Bašić",
}
```

We now know how to write a blueprint to view all books and to create a new book, but how about viewing a single book by its ID.

In the next chapter, we will create a new resource called *Book*, where we will create an action for viewing a single book by its ID and demonstrate using URI templates and parameters.

URI parameters & template

To demonstrate URI parameters and URI template, we will first create a new resource inside our Book resource group.

```
## Book [/books/{book_id}]
```

This resource allows you to view a single book.

In the section heading URI (the text inside the brackets), you can see a parameter {book_id}. The API uses this parameter to retrieve the desired book.

We will now declare the parameters for this resource:

+ Parameters

+ book_id (number) - The ID of the book you want to view.

This means that parameter book_id must be an integer and is required.

Let's create an action for this resource called View a single book which uses the GET HTTP method:

```
### View a single book [GET]
```

This action returns a book in JSON format.

+ Response 200 (application/json)

```
{
  "id": 3,
  "title": "The answer to everything",
  "author": "Mario Bašić",
}
```

That is it! You now know everything needed to create your first API Blueprint. Go and build something!

You can view the complete API Blueprint for this tutorial on [GitHub Gist](#)²⁷⁹.

There are few *advanced* things that have not been mentioned in this tutorial.

Advanced features

This tutorial will cover advanced topics like JSON Schema, attributes, data structures and relation types.

Published at: 12. March, 2017.



View Source Code

Source code for this tutorial is available [here](#)²⁸⁰.

In the previous tutorial, we have covered all the basics. With the knowledge gained from the previous tutorial, you can start writing API blueprints, describing existing APIs or designing new ones.

Once you start doing so, you will begin to notice that a lot of stuff is being retyped, or that you need a finer control of the allowed body content. API Blueprint advanced features cover things like this, which enable you to have more control or reuse existing components which results in saving you time and simplifying the entire blueprint.

²⁷⁹ <https://gist.github.com/mabasic/acd638280aa7b41eabf2d1c737b8cc83>

²⁸⁰ <https://gist.github.com/mabasic/ededf40e476eb76b34bd8b489f380668>

JSON Schema

In our previous tutorial we had this code for creating a new book:

```
### Create a new book [POST]
```

You may create a new book using this action. It takes a JSON object containing a title and an author.

```
+ title (string) - The title of the book.
+ author (string) - Name of the author.
```

```
+ Request (application/json)
```

```
{
  "title": "The answer to everything",
  "author": "Mario Bašić",
}
```

```
+ Response 201 (application/json)
```

```
{
  "id": 3,
  "title": "The answer to everything",
  "author": "Mario Bašić",
}
```

Currently, as you can see in the Request section, we are hard coding an example request body. From this request body, you cannot see which fields are required and of what type they are. You can only guess, or read in the documentation if it says so.

Using the JSON schema to describe the allowed structure of the body content, we can describe the type of each field, which fields are required and their default values. “JSON bodies are typically described with JSON Schema.” To learn more about JSON Schema, click [here](http://json-schema.org/)²⁸¹. Also, this official JSON schema [example](http://json-schema.org/example1.html)²⁸² really explains it all.

JSON schema can be added via an + Schema section.

Now using the JSON schema for the request of our above example, we get this:

²⁸¹<http://json-schema.org/>

²⁸²<http://json-schema.org/example1.html>

```
### Create a new book - JSON schema [POST]
```

You may create a new book using this action. It takes a JSON object containing a title and an author.

```
+ title (string) - The title of the book.
```

```
+ author (string) - Name of the author.
```

```
+ Request (application/json)
```

```
  + Body
```

```
    {
      "title": "The answer to everything",
      "author": "Mario Bašić",
    }
```

```
  + Schema
```

```
    {
      "$schema": "http://json-schema.org/draft-04/schema#",
      "title": "Book",
      "type": "object",
      "properties": {
        "title": {
          "type": "string"
        },
        "author": {
          "type": "string"
        }
      },
      "required": ["title", "author"]
    }
```

```
+ Response 201 (application/json)
```

```
  {
    "id": 3,
    "title": "The answer to everything",
    "author": "Mario Bašić",
  }
```

If you ask me, I think that this is very cool. **Remember**, you also apply the JSON schema to the

response section. Don't forget to add the id field if you do so:

```
"properties": {
  "id": {
    "description": "The unique identifier for a book",
    "type": "integer"
  }
},
```

Attributes aka MSON

Now, if you didn't like the previous chapter where I said that you could use JSON schema to describe the allowed structure of the response and request bodies, you can choose to use MSON (Markdown Syntax for Object Notation). Click [here](#)²⁸³ to find out more.

MSON is an alternative to JSON schema for describing data structures. It was created by the same team that created API blueprint.

“MSON is a plain-text, human and machine readable, description format for describing data structures in common markup formats such as JSON, XML or YAML.”

MSON can be added via an + Attributes section.

Now using MSON for the request of our original example, we get this:

```
### Create a new book - MSON [POST]
```

You may create a new book using this action. It takes a JSON object containing a title and an author.

```
+ title (string) - The title of the book.
+ author (string) - Name of the author.
```

```
+ Request (application/json)
```

```
  + Attributes
```

```
    + title: The answer to everything (required)
    + author: Mario Bašić (required)
```

Now, as you can see, the entire code is much simpler and easier to read than with what we had with the JSON schema.

When the above blueprint is parsed it will have a JSON body and JSON Schema example generated for it from the MSON attributes. *Generated JSON Schema may differ from a hand-written one.*

²⁸³<https://github.com/apiaryio/mson#readme>

Data structures

Until this point, we had only described the request and response body content for a single request. But, what if we have many requests which use the same resource like we do in our Bookstore API blueprint example. Yes, you could copy paste the code, but if you are using MSON you could reference existing data structures from your document.

Data structures are defined in the `# Data Structures` section of your API blueprint. *It is a reserved keyword.*

So for our example, we would have this data structure:

```
# Data Structures

## Book

+ title: The answer to everything (required)
+ author: Mario Bašić (required)
```

And then in the blueprint, we can reference it like this:

```
### Create a new book - MSON + data structure [POST]
```

You may create a new book using this action. It takes a JSON object containing a title and an author.

```
+ title (string) - The title of the book.
+ author (string) - Name of the author.

+ Request (application/json)

    + Attributes (Book)

+ Response 201 (application/json)

    {
        "id": 3,
        "title": "The answer to everything",
        "author": "Mario Bašić",
    }
```

You could apply the same approach to the response section, but you would then need to add `id` to the data structure or create a new one.

Relation types

Ok, so to be quite clear, I personally don't understand this part very well.

To the best of my understanding of this: An action can have a specific meaning regardless of its URI `/books/1` or name `View a book`. If this makes sense to you, good. Please let me know in the comments below.

Relation types are defined in the `+ Relation` section of an action. You can find all the relation types in the [IANA Link Relation Types²⁸⁴](#) document.

So for our example, we would have:

```
### View a single book [GET]
```

This action returns a book in JSON format.

```
+ Relation: self
```

```
+ Response 200 (application/json)
```

```
{
  "id": 3,
  "title": "The answer to everything",
  "author": "Mario Bašić",
}
```

In the IANA Link Relation Types document, relation type `self` has a description: "Conveys an identifier for the link's context."

Now I don't know why anybody would use this, but that is maybe just me. If you have a real world scenario/example on where and why you used this feature, please let me know in the comments below.

This concludes this tutorial. You can view the entire source code for this tutorial on [GitHub Gist²⁸⁵](#).

In this tutorial, we have learned a lot of useful ways of dealing with resources: JSON schema, MSON and data structures. I really like the level of control that JSON schema provides you, but I also like the MSON way because it is readable even to nondevelopers. MSON + data structures provide a very easy and fast way of dealing with resources.

Thank you for reading! This tutorial concludes this course.

²⁸⁴<http://www.iana.org/assignments/link-relations/link-relations.xhtml>

²⁸⁵<https://gist.github.com/mabasic/ededf40e476eb76b34bd8b489f380668>

Database & Eloquent ORM

Learn all about pagination presenters and how to paginate a collection of different models. Also, learn different ways of deleting records.

Paginating a collection of different models

This is most commonly used in the situation where you have to paginate search results which consist of many different models.

Published at: 21. January, 2017.



View Source Code

Source code for this tutorial is available [here](#)²⁸⁶.

To paginate a collection that consists of many different models is a very tricky task, but it can be solved by manually creating a paginator and passing it the collection of models. The task of manually creating a paginator is very poorly documented in the [official documentation](#)²⁸⁷.

In this tutorial, we will implement a search functionality that will work across with two models with pagination. We will accept user input, search the database, retrieve records, merge them in a collection, manually create a paginator and finally display them to the user.

Start

We will create a simple website on which the users can search for desks and chairs. It will be called **Desk & Chair**. Later on, they can maybe order the chair or desk from the seller, by using the item code.

I have prepared the application with migrations, seeders, models and a basic layout. We will now implement the search functionality.

This is the current state of the repository at this moment: [77d3962beef96a2989568c487f6098fea2c48911](#)²⁸⁸.

If you go to /, the application looks like this:

²⁸⁶<https://github.com/laravelista/paginating-a-collection-of-different-models>

²⁸⁷<https://laravel.com/docs/5.3/pagination#manually-creating-a-paginator>

²⁸⁸<https://github.com/laravelista/paginating-a-collection-of-different-models/commit/77d3962beef96a2989568c487f6098fea2c48911>



Desk & Chair

Search for...

Home page

Flow

This is what we want to happen when the user enters a value and clicks on the search button:

1. Validate user input
2. Search & retrieve records from the database
3. Merge records in a collection
4. Paginate the collection
5. Display results

Validation

We will start by validating the input.

Add Request `$request` to the home method parameters in `app/Http/SearchController`. Then, we need to validate the query field which holds the value that the user has entered in the search form. Now the home method looks like this:

```
public function home(Request $request)
{
    $this->validate($request, [
        'query' => 'string',
    ]);

    return view('welcome');
}
```

Save the query value in a variable for later reuse:

```
$query = $request->get('query');
```

If the query is empty, we need to display the welcome view without doing anything else. To do that, we will replace `return view('welcome');` with:

```
if(empty($query)) {
    return view('welcome');
}
```

Now, whenever the query is empty, the method will just return the welcome view.

Database search

We will now search the database for given query. We need to search both desks and chairs tables and save the results in separate variables.

Add this code to search for desks:

```
$desks = \App\Desk::where('name', 'LIKE', '%'.$query.'%')
    ->orWhere('color', 'LIKE', '%'.$query.'%')
    ->orWhere('material', 'LIKE', '%'.$query.'%')
    ->orWhere('description', 'LIKE', '%'.$query.'%')
    ->orWhere('code', 'LIKE', '%'.$query.'%')
    ->get();
```

And, add this code to search for chairs:

```
$chairs = \App\Chair::where('name', 'LIKE', '%'.$query.'%')
    ->orWhere('color', 'LIKE', '%'.$query.'%')
    ->orWhere('material', 'LIKE', '%'.$query.'%')
    ->orWhere('description', 'LIKE', '%'.$query.'%')
    ->orWhere('code', 'LIKE', '%'.$query.'%')
    ->get();
```

Merge records

We will now create an empty collection and merge found desks and chairs into it.

```
$collection = collect([])
    ->merge($desks)
    ->merge($chairs);
```

Create a paginator

To create a paginator there are a couple of things that we need to do. We need to:

- define the number of records that we want to display per page
- provide a total item count of the collection
- provide a current page number
- properly slice the collection

Defining a number of records that we want to display is easy. Just add a variable called `$perPage` and set it to whatever number you want. To demonstrate pagination in this application I will set it to 2.

```
$perPage = 2;
```

We can get the current page number from the request:

```
$page = $request->get('page', 1);
```

If page is not provided in the request it will default to 1.

Now, we can also append a validation rule for page at the top of our method under `$this->validate:`

```
$this->validate($request, [
    'query' => 'string',
    'page' => 'integer',
]);
```

Now that we have `$perPage` and `$page` variables, we can calculate the *offset* with which we will slice the collection. Slicing works like this: it skips *offset* number of items from the collection and takes *perPage* number of following items. To get the `$offset` add this line:

```
$offset = (($page - 1) * $perPage);
```

We are now ready to create a paginator.

```
$results = new \Illuminate\Pagination\LengthAwarePaginator(
    $collection->slice($offset, $perPage)->all(),
    $collection->count(),
    $perPage,
    $page
);
```

Display results

There are a few steps needed to display the results. First, we need to pass the `$results` to the view:

```
return view('welcome')->with(compact('results'));
```

Then go inside `resources/views/welcome.blade.php` and below the `div` tag with the `jumbotron` class, add this code:

```
@if(isset($results) and $results->count() > 0)
    <table class="table">
        <tr>
            <th>Code</th>
            <th>Name</th>
            <th>Color</th>
            <th>Material</th>
            <th>Description</th>
            <th>Type</th>
        </tr>
        @foreach($results as $item)
```

```

        <tr>
            <td>{{ $item->code }}</td>
            <td>{{ $item->name }}</td>
            <td>{{ $item->color }}</td>
            <td>{{ $item->material }}</td>
            <td>{{ $item->description }}</td>
            <td>
                @if($item instanceof App\Desk)
                    Desk
                @elseif($item instanceof App\Chair)
                    Chair
                @endif
            </td>
        </tr>
    @endforeach
</table>

    {{ $results->appends(['query' => Request::get('query')])>links() }}
@endif

```

This code will first check if the `$results` variable is passed to the view and if it is, then it will display the results in a table. The last column in the table displays if the given item is a Desk or a Chair model. At the end it provides pagination with `$results->links()` and passes the query parameter with `->appends(['query' => Request::get('query')])`.

In the situation, if the entered query produces no results, we need to display a message to the user. To do that, add this code below the code that you have just pasted:

```

@if(isset($results) and $results->count() == 0)
    <p class="lead">Nothing found!</p>
@endif

```

This is what the page looks like if you search for something now:

Desk & Chair

Code	Name	Color	Material	Description	Type
de-pinkmix-1	Banging Desk	pink	mixed	It is a heavy pink desk made with wood and glass.	Desk
ch-brownwood-1	Kitchen Chair	brown	wood	It is a normal brown chair made of wood.	Chair

«
1
2
3
»

Search

This is the current state of the repository at this moment: [91f42ad1787ec85cd5d6e76a1c6b98e5ab6a9808](https://github.com/laravelista/paginating-a-collection-of-different-models/commit/91f42ad1787ec85cd5d6e76a1c6b98e5ab6a9808)²⁸⁹.

Job well done! You have completed this tutorial. You can now use this knowledge to implement search functionality in your own application or to display different types of models in a group. Based on the type of model you can now do appropriate actions.

²⁸⁹<https://github.com/laravelista/paginating-a-collection-of-different-models/commit/91f42ad1787ec85cd5d6e76a1c6b98e5ab6a9808>

MySQL ON DELETE CASCADE or SET NULL

Learn how to use referential actions for a foreign key to delete data from child tables when you delete data from a parent table and when is appropriate to set the value to null or leave default.

Published at: 25. August, 2016.

I'm here to tell you about referential actions when defining foreign keys in your database migration files in Laravel.

The reason why I want to talk about this and explain what does this do is because I have been kicked in the ass by it during the refactoring of my website Laravelista.

This is what happened to me

I had some understanding of how cascading works when deleting parent tables and how it reflects on the child tables. The most simple example would be this. When you delete an author and if you have cascade on delete set up for his posts, you would actually delete that author from the authors table and automatically all posts which were written by that author. *This saves you a lot of time.* In your code, you no longer need to first delete the posts from that author and then the author. You just delete the author and automatically the posts are also deleted. If you think about it this makes perfect sense to use everywhere at first. At least that is what I have done for my tutorials and courses. When I deleted a courses it deleted all the tutorials that belonged to that courses. What this comes down to is your business logic and how you want things to happen. For me, this was unacceptable because tutorials are the most important thing for me, not the courses.

So I asked the Internet what other options do I have for this. I found out that you can also use SET NULL instead of CASCADE and what does that do is it sets the value of that foreign key to null, meaning that in my case the tutorial that belonged to the courses that is being deleted now, does not belong to any courses anymore.

Another approach that you could take is to leave out the ON DELETE clause and before deleting a courses manually set the value of the foreign key on tutorials to null. This requires a little bit more code from your part but is still legit.

In the next chapters, I will show you how to set up and use the above mentioned referential actions in your database migration files in Laravel.

CASCADE

To demonstrate when you would use CASCADE referential action I will use the following example. Let's say that we have authors and posts tables. An author can have many posts, while a single post can only belong to a single author.

Now the business rule: *"If the author of the post is deleted, delete all the posts that he has written"*.

To solve this we would create a migration like this:

```
Schema::table('posts', function ($table) {
    // ...
    $table->integer('author_id')->unsigned();
    $table->foreign('author_id')
        ->references('id')->on('authors')
        ->onDelete('cascade');
});
```

SET NULL

To demonstrate SET NULL referential action I will use a different example because it does not make sense to set the author of the post to null. If your business rule requires it so, then you could use it there, but I will use a better example.

Let's say that we now have two tables tutorials and courses. One tutorial can belong to only one courses and one courses can have many tutorials.

Now the business rule: *"If the courses is deleted, orphan the tutorials that belonged to that courses"*. What I wanted to say here is to set the course_id value to null, meaning that the tutorial that has previously belonged to a courses, now no longer belongs to any courses and is independent.

To solve this we would create a migration like this:

```
Schema::table('tutorials', function ($table) {
    // ...
    $table->integer('course_id')->unsigned();
    $table->foreign('course_id')
        ->references('id')->on('courses')
        ->onDelete('set null');
});
```

I hope that I have managed to bring closer this topic to you and that you will take it into consideration the next time you write a database migration file.

Dropping a composite primary key on a pivot table

Funny story. If you have found yourself in the situation where you are trying to drop a composite primary key and keep receiving some invalid constraint error then this is the tutorial for you.

Published at: 29. August, 2016.

With this tutorial, I hope to save you some time trying to solve this issue if you ever come by it.

The Issue

I was working on a Laravel 5.2 project and writing migrations for three new tables when I noticed that I have forgotten to create a primary key in a pivot table related to `clients` and `locations` called `client_location`. This is the schema for that table:

```
Schema::create('client_location', function(Blueprint $table) {  
    $table->integer('client_id')->unsigned();  
    $table->foreign('client_id')  
        ->references('id')->on('clients')  
        ->onDelete('cascade');  
  
    $table->integer('location_id')->unsigned();  
    $table->foreign('location_id')  
        ->references('id')->on('locations')  
        ->onDelete('cascade');  
});
```

As you can see, no primary key. What I could have done is just add `$table->increments('id');` primary key and be done with it, but then I would be able to have the same client at the same location many times recorded in the database and that is not good in my case. I only need to have a single record for this many to many relationship. To do that I need a composite primary key. No problem.

I wrote a new migration where in the up method I created that key:

```
Schema::table('client_location', function(Blueprint $table) {
    $table->primary(['client_id', 'location_id']);
});
```

But as many of you already know, you have to write the down method that does the opposite of what you have done in the up method so that you can easily migrate or rollback the database migrations.

My first attempt was very straight forward:

```
Schema::table('client_location', function(Blueprint $table) {
    $table->dropPrimary();
});
```

The way I tested that this works is I ran `php artisan migrate` and then followed by `php artisan migrate:rollback`. If the two commands were successful great, but they weren't. I received an error saying something about invalid constraint on some foreign or something like that. The error is very generic and after searching for the answer on the Internet I came up short.

Finding the solution

Then I have done what every other sane developer would do, throw a bunch of different code at it, trying to solve it:

```
Schema::table('client_location', function(Blueprint $table) {
    $table->dropPrimary(['client_id', 'location_id']);
});
```

// and

```
Schema::table('client_location', function(Blueprint $table) {
    $table->dropPrimary('client_id');
    $table->dropPrimary('location_id');
});
```

But every time the same error occurred. Great.

Finding the solution 2.0

At this point “shit got real”. I have spent several hours trying to solve this and because of this bug, I could not move on to actual coding. Luckily for me, on my PC I already had [MySQL Workbench](http://www.mysql.com/products/workbench/)²⁹⁰

²⁹⁰<http://www.mysql.com/products/workbench/>

installed. I connected to the Homestead database that I was working on and carefully inspected that tables' changes during the migration. I have found that because the primary key is constructed from two columns that have the foreign key constraint on them I was unable to drop the primary key for some reason. It still makes little sense to me, but OK. Take it as it is, I modified my down method a bit and came up with working code:

```
Schema::table('client_location', function(Blueprint $table) {
    $table->dropForeign('client_location_client_id_foreign');
    $table->dropForeign('client_location_location_id_foreign');
    $table->dropPrimary();
});
```

The Solution

First I drop the foreign key constraint and then I drop the primary key. Voila, it works! Because I dropped the foreign key constraints just to drop the primary key I had to add the foreign keys back after I deleted the primary key like so:

```
Schema::table('client_location', function(Blueprint $table) {
    $table->foreign('client_id')
        ->references('id')->on('clients')
        ->onDelete('cascade');
    $table->foreign('location_id')
        ->references('id')->on('locations')
        ->onDelete('cascade');
});
```

Now everything works and I can move on to actual coding the application. I really don't know why I could not just drop the primary key in this situation. This really is an edge situation in which you almost never come by, but it happened to me.

Understanding Pagination Presenters

Pagination presenters enable you to implement your own pagination component design. You can extend the default Bootstrap presenter and modify it to suit your needs or you can create a truly unique pagination component.

Published at: 09. August, 2016.



View Source Code

Source code for this tutorial is available [here](#)²⁹¹.

At the time of writing this tutorial, I was in the process of refactoring the backend for Laravelista. I have bought a nice Material admin template with a lot of cool stuff including a custom pagination component.

It took me some time at first to figure out what do I have to do to use my own pagination component instead of the one that comes with Laravel out-of-the-box. First, misleading thing was the official Laravel documentation for Pagination specifically the part about [Manually Creating A Paginator](#)²⁹².

Trust me, in most cases, the chances are that you really want to create a pagination presenter, not a paginator. In this tutorial, I will show you different ways how you can create a paginator presenter and use it on your website.

Default Pagination

To use a paginator on an eloquent model collection there are a few things that you have to do. First, you have to call the `paginate(25)` method on the model

```
$users = \App\User::paginate(10);
```

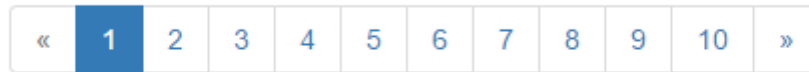
and then inside the view template you have to call `->links()` method on the model collection

²⁹¹<https://github.com/laravelista/understanding-custom-pagination-presenters>

²⁹²<https://laravel.com/docs/5.2/pagination#manually-creating-a-paginator>

```
{{ $users->links() }}
```

This will generate a [Bootstrap](#)²⁹³ compatible pagination component which looks like this:



Default Paginator

To view the presenter in charge of that pagination component, look for `Illuminate\Pagination\BootstrapThreePresenter` class.

This is the current state of the repository at this moment [5d7d8bade323abf6aec374389479e4a749e0c1d0](#)²⁹⁴.

Extend Bootstrap Presenter

You would extend the Bootstrap presenter if you are happy with it, but just want to modify small bits here and there. Let's say that instead of symbols » and «, you want to display letters P for previous and N for next.

This is a really silly example but stay with me on this one because I will show you how easily this can be done.

First we create a new file `app/Pagination/CustomPaginationPresenter.php` and paste the following inside:

```
<?php namespace App\Pagination;

use Illuminate\Pagination\BootstrapThreePresenter;

class CustomPaginationPresenter extends BootstrapThreePresenter
{
    //
}
```

We have now created a new class called `CustomPaginationPresenter` which extends the default `BootstrapThreePresenter` class.

To use our newly created presenter class instead of the default one, we have to pass it as a parameter to the `links()` method mentioned in the chapter before.

²⁹³<http://getbootstrap.com/components/#pagination>

²⁹⁴<https://github.com/laravelista/understanding-custom-pagination-presenters/commit/5d7d8bade323abf6aec374389479e4a749e0c1d0>


```
{{ $users->links(new \App\Pagination\CustomPaginationPresenter($users)) }}
```

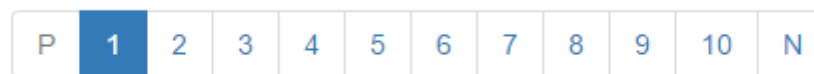
If you would to check it now in the browser, you would see no difference because we have just extended the default presenter and have done nothing with it.

Now we want to display letters P for previous and N for next. To do that we have to override method render from the extended presenter class. Add this code to our CustomPaginationPresenter class:

```
public function render()
{
    if ($this->hasPages()) {
        return new HtmlString(sprintf(
            '<ul class="pagination">%s %s %s</ul>',
            $this->getPreviousButton('P'),
            $this->getLinks(),
            $this->getNextButton('N')
        ));
    }

    return '';
}
```

We have just copied the render method from BootstrapThreePresenter class and made little modifications to it. If you inspect the BootstrapThreePresenter class you will see that both getPreviousButton and getNextButton methods accept a text parameter. This means that we can easily change the pagination component to have letter P for previous and letter N for next, as you can see here:



Extended Presenter

This is the current state of the repository at this moment [5fd73d7d711667b40fda075c77e50baf0ebf2d62](https://github.com/laravelista/understanding-custom-pagination-presenters/commit/5fd73d7d711667b40fda075c77e50baf0ebf2d62)²⁹⁵.

If you are looking to modify the default presenter look no further, but if you want to create a unique presenter then read on.

I myself have not yet needed to write a presenter from scratch since almost all of my projects are based on Bootstrap.

²⁹⁵<https://github.com/laravelista/understanding-custom-pagination-presenters/commit/5fd73d7d711667b40fda075c77e50baf0ebf2d62>

Create a New Presenter

If you still want to write a Presenter class, let me give you some pointers and a head start. If you have reached this point, I sure hope that you know what you are doing. That being said, let's dive straight in. This is the bare minimum for a Presenter class:

```
<?php namespace App\Pagination;

use Illuminate\Support\HtmlString;
use Illuminate\Contracts\Pagination\Paginator as PaginatorContract;
use Illuminate\Contracts\Pagination\Presenter as PresenterContract;

class BlankPaginationPresenter implements PresenterContract
{
    /**
     * The paginator implementation.
     *
     * @var \Illuminate\Contracts\Pagination\Paginator
     */
    protected $paginator;

    /**
     * Create a new Bootstrap presenter instance.
     *
     * @param \Illuminate\Contracts\Pagination\Paginator $paginator
     * @return void
     */
    public function __construct(PaginatorContract $paginator)
    {
        $this->paginator = $paginator;
    }

    /**
     * Render the given paginator.
     *
     * @return \Illuminate\Contracts\Support\Htmlable|string
     */
    public function render()
    {
        return new HtmlString("<p>Replace this line with your paginator.</p>");
    }
}
```

```

/**
 * Determine if the underlying paginator being presented has pages to show.
 *
 * @return bool
 */
public function hasPages()
{
    return $this->paginator->hasPages();
}
}

```

To test that it works use:

```
{{ $users->links(new \App\Pagination\BlankPaginationPresenter($users)) }}
```

You should see paragraph Replace this line with your paginator. in your browser.

The main method here is the render method. That method is in charge for actual displaying the pagination component.

This is the current state of the repository at this moment [f4ea4c2b559a79a348e3e3c0c2f77c4ce1539c4e](https://github.com/laravelista/understanding-custom-pagination-presenters/commit/f4ea4c2b559a79a348e3e3c0c2f77c4ce1539c4e)²⁹⁶.

This concludes this tutorial.

²⁹⁶<https://github.com/laravelista/understanding-custom-pagination-presenters/commit/f4ea4c2b559a79a348e3e3c0c2f77c4ce1539c4e>

Testing

Testing with Codeception on Laravel and Lumen.

Laravel 5 with Codeception

In this tutorial, I will show you how to replace default Laravel 5 testing suites with Codeception only.

Published at: 04. June, 2017.

Continuing on my post from 2014: **Getting started with Codeception**. In this tutorial, I will show you how to install Codeception to work with Laravel 5 and remove the default PHPUnit testing suite.

Basically what we want to do is to replace the default PHPUnit testing suite with Codeception. I have done this for this website (Laravelista) because I find Codeception to be very easy to use and write tests.

Let's begin.

This is the current state of the repository at this moment [a1f918027ae3a50f82fbcf68735bb68840cbefff](https://github.com/laravelista/laravel-codeception/commit/a1f918027ae3a50f82fbcf68735bb68840cbefff)²⁹⁷.

Install Codeception

We will install Codeception as a development dependency using Composer from the command line:

```
composer require codeception/codeception --dev
```

Output:

```
$ composer require codeception/codeception --dev
Using version ^2.3 for codeception/codeception
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing symfony/dom-crawler (v3.3.0)
  Downloading: 100%

- Installing symfony/browser-kit (v3.3.0)
  Downloading: 100%
```

²⁹⁷ <https://github.com/laravelista/laravel-codeception/commit/a1f918027ae3a50f82fbcf68735bb68840cbefff>

- Installing stecman/symfony-console-completion (0.7.0)
Downloading: 100%
- Installing psr/http-message (1.0.1)
Downloading: 100%
- Installing guzzlehttp/psr7 (1.4.2)
Loading from cache
- Installing guzzlehttp/promises (v1.3.1)
Loading from cache
- Installing guzzlehttp/guzzle (6.2.3)
Loading from cache
- Installing facebook/webdriver (1.4.1)
Downloading: 100%
- Installing behat/gherkin (v4.4.5)
Loading from cache
- Installing codeception/codeception (2.3.3)
Downloading: 100%

This is the current state of the repository at this moment [4aa82f5956bd6fd7428c0ad3b8443ba53cb17839](https://github.com/laravelista/laravel-codeception/commit/4aa82f5956bd6fd7428c0ad3b8443ba53cb17839)²⁹⁸.

Default testing suite cleanup

First, we will remove the "phpunit/phpunit": "~5.7" dependency from composer.json. Then run `composer update` to reflect the change.

Now from the application root delete the file `phpunit.xml`, and delete the directory called `tests`.

This is the current state of the repository at this moment [f3adbd705f06bc505b7381634f6ad936266272bc](https://github.com/laravelista/laravel-codeception/commit/f3adbd705f06bc505b7381634f6ad936266272bc)²⁹⁹.

Bootstrapping Codeception

To bootstrap (finish installing/integrate) Codeception we need to execute this command:

²⁹⁸ <https://github.com/laravelista/laravel-codeception/commit/4aa82f5956bd6fd7428c0ad3b8443ba53cb17839>

²⁹⁹ <https://github.com/laravelista/laravel-codeception/commit/f3adbd705f06bc505b7381634f6ad936266272bc>

```
vendor/bin/codecept bootstrap
```

Output:

```
$ vendor/bin/codecept bootstrap
Bootstrapping Codeception

File codeception.yml created      <- global configuration
> Unit helper has been created in tests/_support\Helper
> UnitTester actor has been created in tests/_support
> Actions have been loaded
tests/unit created                <- unit tests
tests/unit.suite.yml written      <- unit tests suite configuration
> Functional helper has been created in tests/_support\Helper
> FunctionalTester actor has been created in tests/_support
> Actions have been loaded
tests/functional created          <- functional tests
tests/functional.suite.yml written <- functional tests suite configuration
> Acceptance helper has been created in tests/_support\Helper
> AcceptanceTester actor has been created in tests/_support
> Actions have been loaded
tests/acceptance created         <- acceptance tests
tests/acceptance.suite.yml written <- acceptance tests suite configuration
---
```

Codeception is installed **for** acceptance, functional, and unit testing

As you can see Codeception is installed for acceptance, functional, and unit testing. Also, the tests directory is recreated with the Codeception folder structure. You can change the structure if needed, but I find this to be the best option.

This is the current state of the repository at this moment [3b10d8383fc7fc69c9e305a16bc2ea28647ace76](https://github.com/laravelista/laravel-codeception/commit/3b10d8383fc7fc69c9e305a16bc2ea28647ace76)³⁰⁰.

Installing Laravel5 module

For this tutorial, I will write a functional test that simply checks if the welcome page returns 200 HTTP code and if there is text on page Laravel.

To better integrate Laravel with Codeception there is a module called [Laravel5](http://codeception.com/docs/modules/Laravel5)³⁰¹. It enables you to use **methods** like:

³⁰⁰<https://github.com/laravelista/laravel-codeception/commit/3b10d8383fc7fc69c9e305a16bc2ea28647ace76>

³⁰¹<http://codeception.com/docs/modules/Laravel5>

- `amLoggedAs`
- `amOnPage`
- `amOnRoute`
- `click`
- `dontSee`
- `fillField`
- `have`
- `see`
- `seeCurrentRouteIs`
- `seeInSession`
- `seeLink`
- `seePageNotFound`
- `seeRecord`

And many other useful helper methods.

Other than being able to use helpful methods, it comes with a variety of **configuration options** that you can tweak:

- `cleanup`
- **`run_database_migrations`**
- `database_migrations_path`
- `run_database_seeder`
- `database_seeder_class`
- **`environment_file`**
- `bootstrap`
- `root`
- `packages`
- `vendor_dir`
- `disable_exception_handling`
- `disable_middleware`
- `disable_events`
- `disable_model_events`
- `url`

I have marked two options that I use the most.

This module is to be used for functional testing only. That means that we need to enable it in `tests/functional.suite.yml`. Make that file look like this:


```
actor: FunctionalTester
modules:
  enabled:
    - Laravel5
    - \Helper\Functional
```

Now run from the command line:

```
vendor/bin/codecept build
```

Awesome! Installation complete.

Optional: If you want to change the configuration options for the module you can do that like so:

```
actor: FunctionalTester
modules:
  enabled:
    - Laravel5:
        environment_file: .env.testing
        run_database_migrations: true
    - \Helper\Functional
```

This is the current state of the repository at this moment [aaecc75e463ac1e8273a467878cd491bc5d979f6](https://github.com/laravelista/laravel-codeception/commit/aaecc75e463ac1e8273a467878cd491bc5d979f6)³⁰².

Your first functional Codeception test

To create a new functional test using the cept format, run this command:

```
vendor/bin/codecept generate:cept functional WelcomePage
```

Output:

```
$ vendor/bin/codecept generate:cept functional WelcomePage
Test was created in C:\repos\laravel-codeception\tests\functional\WelcomePageCep\
t.php
```

You can find the test in `\tests\functional\WelcomePageCept.php`. Now modify the file so that it looks like this:

³⁰²<https://github.com/laravelista/laravel-codeception/commit/aaecc75e463ac1e8273a467878cd491bc5d979f6>

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('view the welcome page');

$I->amOnPage('/');
$I->seeResponseCodeIs(200);
$I->see('Laravel');
```

We navigate to /, check if the response code is 200, and verify that we see text Laravel on the page.

Before we run the test there is one more thing to modify. Because our sample application does not use a database we need to modify our database values in the .env file:

```
DB_CONNECTION=sqlite
DB_DATABASE=:memory:
```

Remove all other keys starting with DB_.

Now, to run the test execute:

```
vendor/bin/codecept run functional
```

Output:

```
$ vendor/bin/codecept run functional
Codeception PHP Testing Framework v2.3.3
Powered by PHPUnit 6.2.1 by Sebastian Bergmann and contributors.
```

```
Functional Tests (1) ----- \
-----
+ WelcomePageCept: View the welcome page (0.04s)
----- \
-----
```

```
Time: 514 ms, Memory: 16.00MB
```

```
OK (1 test, 2 assertions)
```

Success!

This is the current state of the repository at this moment [1c14715f10b506a46c076677eaf4069cbdd8d13](https://github.com/laravelista/laravel-codeception/commit/1c14715f10b506a46c076677eaf4069cbdd8d13)³⁰³.

You are now ready to start writing tests for your Laravel 5 application.

³⁰³<https://github.com/laravelista/laravel-codeception/commit/1c14715f10b506a46c076677eaf4069cbdd8d13>

Lumen 5 with Codeception

In this tutorial, I will show you how to replace the default Lumen 5 testing suite with Codeception.

Published at: 11. June, 2017.

In my previous tutorial about [Laravel 5 with Codeception](#) I have shown you how to replace the default Laravel 5 testing suite (PHPUnit) with Codeception. In this tutorial, I will show you how to replace the default Lumen 5 testing suite (PHPUnit) with Codeception.

It has been a long time since I have dealt with Lumen. In my tutorial [JSON Web Token Authentication for Lumen REBOOT](#) I have made a conclusion about Lumen:

If you need a full blown API you are better off with Laravel. If you need a simple API that does a thing or two, you will be better off with Lumen, but if you get to the point where you need JWT in Lumen you are probably off track and doing too much in it.

I must admit, since then I have not worked with Lumen so I thought that this tutorial will be a nice introduction back to Lumen.

Let's begin.

This is the current state of the repository at this moment [618d292f2651b1557e3161e1e6d1eb22ddb99f](https://github.com/laravelista/lumen-codeception/commit/618d292f2651b1557e3161e1e6d1eb22ddb99f)³⁰⁴.

Install Codeception

We will install Codeception as a development dependency using Composer from the command line:

```
composer require codeception/codeception --dev
```

Output:

³⁰⁴<https://github.com/laravelista/lumen-codeception/commit/618d292f2651b1557e3161e1e6d1eb22ddb99f>

```

$ composer require codeception/codeception --dev
Using version ^2.3 for codeception/codeception
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing symfony/dom-crawler (v3.3.2)
  Loading from cache

- Installing symfony/css-selector (v3.3.2)
  Loading from cache

- Installing symfony/browser-kit (v3.3.2)
  Loading from cache

- Installing stecman/symfony-console-completion (0.7.0)
  Loading from cache

- Installing psr/http-message (1.0.1)
  Loading from cache

- Installing guzzlehttp/psr7 (1.4.2)
  Loading from cache

- Installing guzzlehttp/promises (v1.3.1)
  Loading from cache

- Installing guzzlehttp/guzzle (6.2.3)
  Loading from cache

- Installing facebook/webdriver (1.4.1)
  Loading from cache

- Installing behat/gherkin (v4.4.5)
  Downloading: 100%

- Installing codeception/codeception (2.3.3)
  Loading from cache

```

This is the current state of the repository at this moment [61c7a8cc07f3db39605be8a342884b88082accea](https://github.com/laravelista/laravel-codeception/commit/61c7a8cc07f3db39605be8a342884b88082accea)³⁰⁵.

³⁰⁵<https://github.com/laravelista/laravel-codeception/commit/61c7a8cc07f3db39605be8a342884b88082accea>

Default testing suite cleanup

First, we will remove the "phpunit/phpunit": "~5.0" dependency from `composer.json`. Then run `composer update` to reflect the change.

Now from the application root delete the file `phpunit.xml`, and delete the directory called `tests`.

This is the current state of the repository at this moment [6f79e76d3b8f08eec49ec6ddf8a989d7eab7d428](https://github.com/laravelista/laravel-codeception/commit/6f79e76d3b8f08eec49ec6ddf8a989d7eab7d428)³⁰⁶.

Bootstrapping Codeception

To bootstrap (finish installing/integrate) Codeception we need to execute this command:

```
vendor/bin/codecept bootstrap
```

Output:

```
$ vendor/bin/codecept bootstrap
Bootstrapping Codeception

File codeception.yml created      <- global configuration
> Unit helper has been created in tests/_support\Helper
> UnitTester actor has been created in tests/_support
> Actions have been loaded
tests/unit created               <- unit tests
tests/unit.suite.yml written     <- unit tests suite configuration
> Functional helper has been created in tests/_support\Helper
> FunctionalTester actor has been created in tests/_support
> Actions have been loaded
tests/functional created         <- functional tests
tests/functional.suite.yml written <- functional tests suite configuration
> Acceptance helper has been created in tests/_support\Helper
> AcceptanceTester actor has been created in tests/_support
> Actions have been loaded
tests/acceptance created         <- acceptance tests
tests/acceptance.suite.yml written <- acceptance tests suite configuration
---
```

Codeception is installed **for** acceptance, functional, and unit testing

³⁰⁶<https://github.com/laravelista/laravel-codeception/commit/6f79e76d3b8f08eec49ec6ddf8a989d7eab7d428>

As you can see Codeception is installed for acceptance, functional, and unit testing. Also, the tests directory is recreated with the Codeception folder structure. You can change the structure if needed, but I find this to be the best option.

This is the current state of the repository at this moment [8819e8d47e851e092bc459237a9ce552d7571a12](https://github.com/laravelista/laravel-codeception/commit/8819e8d47e851e092bc459237a9ce552d7571a12)³⁰⁷.

Installing Lumen module

For this tutorial, I will write a functional test that simply checks if the page / returns 200 HTTP code and if there is text on page Lumen.

To better integrate Lumen with Codeception there is a module called [Lumen](#)³⁰⁸. It enables you to use **methods** like:

- amLoggedInAs
- amOnPage
- amOnRoute
- click
- dontSee
- fillField
- have
- see
- seeLink
- seePageNotFound
- seeRecord

And many other useful helper methods.

Other than being able to use helpful methods, it comes with a variety of **configuration options** that you can tweak:

- cleanup
- bootstrap
- root
- packages
- url

This module is to be used for functional testing only. That means that we need to enable it in tests/functional.suite.yml. Make that file look like this:

³⁰⁷ <https://github.com/laravelista/laravel-codeception/commit/8819e8d47e851e092bc459237a9ce552d7571a12>

³⁰⁸ <http://codeception.com/docs/modules/Lumen>

```
actor: FunctionalTester
modules:
  enabled:
    - Lumen
    - \Helper\Functional
```

Now run from the command line:

```
vendor/bin/codecept build
```

Awesome! Installation complete.

Optional: If you want to change the configuration options for the module you can do that like so:

```
actor: FunctionalTester
modules:
  enabled:
    - Lumen:
        cleanup: false
    - \Helper\Functional
```

This is the current state of the repository at this moment [fe222ecae5d97af7d52bec12567cddadf9adf4a7](https://github.com/laravelista/laravel-codeception/commit/fe222ecae5d97af7d52bec12567cddadf9adf4a7)³⁰⁹.

Your first functional Codeception test

To create a new functional test using the cept format, run this command:

```
vendor/bin/codecept generate:cept functional WelcomePage
```

Output:

```
$ vendor/bin/codecept generate:cept functional WelcomePage
Test was created in C:\repos\lumen-codeception\tests\functional\WelcomePageCept.\
php
```

You can find the test in `\tests\functional\WelcomePageCept.php`. Now modify the file so that it looks like this:

³⁰⁹ <https://github.com/laravelista/laravel-codeception/commit/fe222ecae5d97af7d52bec12567cddadf9adf4a7>

```
<?php
$I = new FunctionalTester($scenario);
$I->wantTo('view the welcome page');

$I->amOnPage('/');
$I->seeResponseCodeIs(200);
$I->see('Lumen');
```

We navigate to /, check if the response code is 200, and verify that we see text Lumen on the page.

Before we run the test there is one more thing to modify. Because our sample application does not use a database we need to modify our database values in the .env file:

```
DB_CONNECTION=sqlite
DB_DATABASE=:memory:
```

Remove all other keys starting with DB_.

Now, to run the test execute:

```
vendor/bin/codecept run functional
```

Output:

```
$ vendor/bin/codecept run functional
Codeception PHP Testing Framework v2.3.3
Powered by PHPUnit 6.2.1 by Sebastian Bergmann and contributors.
```

```
Functional Tests (1) ----- \
-----
+ WelcomePageCept: View the welcome page (0.07s)
----- \
-----
```

```
Time: 538 ms, Memory: 12.00MB
```

```
OK (1 test, 2 assertions)
```

Success!

This is the current state of the repository at this moment [ec84afdab710e6496187e0b601dae33ec193b141](https://github.com/laravelista/laravel-codeception/commit/ec84afdab710e6496187e0b601dae33ec193b141)³¹⁰.

You are now ready to start writing tests for your Lumen 5 application.

³¹⁰<https://github.com/laravelista/laravel-codeception/commit/ec84afdab710e6496187e0b601dae33ec193b141>

Deployment

Learn everything about Laravel Envoy with practical examples. There is also a tutorial on deploying a Laravel app from GitHub to Heroku.

Deploy Laravel with Envoy

Probably the easiest way to write a deployment script for your Laravel application.

Published at: 29. January, 2017.



View Source Code

Source code for this tutorial is available [here](#)³¹¹.

[Laravel Envoy](#)³¹² is a tool that I have been personally using for a long time now. It is very easy to get started with it because of its familiar Blade syntax. As you already know, Blade is used as a templating engine for Views in Laravel. So if you have used Laravel Views before, you will know how the Blade syntax works.

Don't worry, even if you are new to it, it is very easy to get started.

With Envoy, you can do all kinds of stuff on the remote server. You can even use it to control a local machine. The only limitation of it is that it only works on Mac and Linux operating systems.

In this tutorial, I will show you how to quickly write a deployment task for Laravel, and then finally how to split that single task into multiple smaller reusable tasks with a story.

Installation

If you are on a Mac or a Linux operating system, just run this command to install Envoy globally:

```
composer global require "laravel/envoy=~1.0"
```

If you are on Windows OS as I am, don't panic! There is a way. The chances are that you are probably using [Laravel Homestead](#)³¹³ for your development environment. Great! Laravel Homestead runs on Linux, and it already comes with Laravel Envoy preinstalled!

Boot up your Homestead machine and SSH into it.

To get started with Laravel Homestead on Windows, read my free Course on that topic:
[Laravel on Windows with Homestead](#).

Run envoy from inside the VM, to check that it is installed. You should see a line like this:

³¹¹<https://gist.github.com/mabasic/5d114addbeb1422161de>

³¹²<https://github.com/laravel/envoy>

³¹³<https://laravel.com/docs/5.3/homestead>

```
vagrant@homestead:~$ envoy
Laravel Envoy 1.3.2
```

This means that you have Envoy installed and are ready to continue with the Tutorial.

Getting started

All Envoy tasks for your application should be defined in `Envoy.blade.php` file in the root of your project. You can create that file manually or you can use `envoy init <host>` command. I suggest using the command because it gives you a sample of how the file should look like.

First create/navigate to the root of the project on which you want to use Envoy. Then, type `envoy init "user@yourdomain.com -p 1234"`. This will create a new file `Envoy.blade.php` in the current directory with these defaults:

```
@servers([ 'web' => 'user@yourdomain.com -p 1234' ])

@task('deploy')
    cd /path/to/site
    git pull origin master
@endtask
```

As you can see, for the host we have entered a *user*, *domain* and a *port*. If the port is omitted, the default port for SSH is used 22. Also, if you omit the *user*, the default user root is used. So, for example, if the user you are using to connect to your server is called root and you have the default port for SSH, then you can just enter `envoy init yourdomain.com`.

If you are executing Envoy on the server on which you want to do changes, you can specify the server's IP address as `127.0.0.1`. This will execute tasks on the local server.

There is a server defined, called `web` and a task called `deploy`. Inside the task, each line is executed after the previous command completes. First, the task navigates to the directory where the application is located on the server and then, in the second line it uses git to pull the latest changes from the repository.

While this is fine for a simple PHP application, our imaginary Laravel application requires a bit more work :)

Writing a deployment task

To deploy our Laravel application we will need to:

- navigate to the application root directory
- put the application in maintenance mode
- pull latest changes from the repository
- install/update Composer dependencies
- migrate the database (only if you use a database)
- clear cache (optional)
- refresh routes cache (optional)
- bring back the application from maintenance mode

Let's change the deploy task a bit so that it looks like this:

```
@task('deploy')
    cd /path/to/site
    php artisan down
    git pull origin
    composer install --prefer-dist --no-interaction --no-dev
    php artisan migrate --force
    php artisan cache:clear
    php artisan route:cache
    php artisan up
@endtask
```

In these few lines, we have everything needed to deploy/update our application on the production server.

To run this task enter this command:

```
envoy run deploy
```

Writing a story

Most of the time, when I need to do something quickly I use the above approach. Just write the lines you need and execute. Great! But, if you are working on something more complex you usually need to do a couple of things.

The first step, is to break down a task into smaller separate tasks. Something like this:

```

@task('down')
    cd /path/to/site
    php artisan down
@endtask

@task('git')
    cd /path/to/site
    git pull origin
@endtask

@task('composer')
    cd /path/to/site
    composer install --prefer-dist --no-interaction --no-dev
@endtask

@task('migrate')
    cd /path/to/site
    php artisan migrate --force
@endtask

@task('cache')
    cd /path/to/site
    php artisan cache:clear
@endtask

@task('routes')
    cd /path/to/site
    php artisan route:cache
@endtask

@task('up')
    cd /path/to/site
    php artisan up
@endtask

```

Now, notice how we repeat `cd /path/to/site` in each task. What if we want to change the path to our application later on? We can easily solve this by adding a `@setup` section to our file:

```

@setup
    $path = "/path/to/site";
@endsetup

```

And replacing the `cd /path/to/site` in our tasks to `cd {{ $path }}`.

Finally, we can write a story now. At the end of the file, add this code:

```
@story('deploy')
    down
    git
    composer
    migrate
    cache
    routes
    up
@endstory
```

Be sure to remove the task called `deploy`, in order for this story to work.

You can now see how clean and readable it is to understand what goes on when we deploy our application. You can run the story the same way as tasks:

```
envoy run deploy
```

Or you can even run each task separately using `envoy run git`, `envoy run cache`...

You can view the entire code for this tutorial [here](#)³¹⁴.

Learn more

There are a couple of things that I have not covered in this tutorial, but Envoy supports them:

- including PHP files
- command variables
- `if` statements and loops
- multiple servers
- parallel execution
- confirming task execution
- notifications (slack)

You can read all about them in the [official documentation](#)³¹⁵ or wait for the next tutorial where I will cover them.

³¹⁴<https://gist.github.com/mabasic/5d114addbeb1422161de>

³¹⁵<https://laravel.com/docs/5.3/envoy>

Advanced Laravel Envoy

Continuing from the previous tutorial, we will learn about advanced Laravel Envoy features, how to use them and why.

Published at: 04. February, 2017.

In the previous tutorial [Deploy Laravel with Envoy](#), I have shown you how to deploy a Laravel application using Envoy. The whole process was kept very simple for the purpose of the tutorial. But, in real life there usually is a thing or two extra that you might want to take care of while deploying.

We will go over these Envoy features:

- including PHP files
- command variables
- if statements and loops
- multiple servers
- parallel execution
- confirming task execution
- cleanup
- notifications (slack)

Do you need to know this to use Envoy effectively? No. But, knowing that these features exist and how to use them, may come handy at some point.

Let's start!

Including PHP files

As we already know from the previous tutorial, we can use the `@setup` directive to execute some PHP code before running the tasks.

```
@setup
    $now = new DateTime();
@endsetup
```

But, Envoy allows us to go even a step further and require other PHP files that we can use in the `@setup` directive.

You can include `vendor/autoload.php` file and then get access to any installed PHP package in your application. (`vendor/autoload.php` is only available if you are using Composer to manage your dependencies. If you are working on a Laravel project, then you are all set to go. If not, be sure to start using Composer.)

```
@include('vendor/autoload.php')

@setup
    $date = Carbon\Carbon::now();
@endsetup
```

In this example, I have used the `nesbot/carbon` package to get the current `DateTime`. The example is trivial but imagine the possibilities.

You can now use the `$date` variable in your tasks like so:

```
@task('date')
    echo {{ $date }}
@endtask
```

Command Variables

This is something that I find very useful. You can pass parameters to an Envoy task from the command line.

First, setup your Envoy task:

```
@setup
    $environment = isset($env) ? $env : "testing";
@endsetup

@task('env')
    echo {{ $environment }}
@endtask
```

First, run the task as is:


```
envoy run env
```

You will get testing.

Then, run the command with env parameter set:

```
envoy run env --env=local
```

You will get local in the response. How awesome is that!

if statements and loops

Yes, Envoy supports them. You can execute commands depending on the presence of the command line parameter.

First, write your task:

```
@task('env', ['on' => 'local'])
    @if ($env)
        echo {{ $env }}
    @else
        echo testing
    @endif
@endtask
```

Then you can call the task using `envoy run env`, and you will get testing in response. Or, you can call the task using `envoy run env --env=local`, and you will get local in the response.

This is very useful when you want to deploy but want to skip migrating the database: (example taken from the previous tutorial)

```
@story('deploy')
    down
    git
    composer
    @if(!$skip_migrate)
        migrate
    @endif
    cache
    routes
    up
@endstory
```

Then, to skip migrating the database you can execute the command:

```
envoy run deploy --skip_migrate
```

Loops

Another cool thing that Envoy can do, is that you can use loops in your tasks.

```
@setup
    $loop = isset($loop) ? $loop : 1;
@endsetup

@task('for')
    @for($i = 0; $i <= $loop; $i++)
        echo 'loop number' {{ $i }}
    @endfor
@endtask
```

If you run it with `envoy run for`, you will get:

```
loop number 0
loop number 1
```

Or, you can pass it the loop parameter like so `envoy run for --loop=30`.

Again, the example is trivial, but imagine what you can do with this knowledge.

Multiple servers

This is a feature that I personally don't use as much as I would like to. Basically, you can define multiple servers in your `@servers` declaration and then define on which server a task is to be run.

```
@servers(['web' => 'user@yourdomain.com -p 1234', 'local' => '127.0.0.1'])

@task('deploy', ['on' => ['web', 'local']])
    cd /path/to/site
    php artisan down
    git pull origin
    composer install --prefer-dist --no-interaction --no-dev
    php artisan migrate --force
    php artisan up
@endtask

@task('test', ['on' => 'local'])
    echo {{ $date }}
@endtask
```

Parallel execution

In the previous example, if you run `envoy run deploy`, the task `deploy` will first run on the web server and then on the `local` server. To enable parallel execution you need to add `parallel` option to the task declaration:

```
@task('deploy', ['on' => ['web', 'local'], 'parallel' => true])
    cd /path/to/site
    php artisan down
    git pull origin
    composer install --prefer-dist --no-interaction --no-dev
    php artisan migrate --force
    php artisan up
@endtask
```

Now, the task will run simultaneously on both servers.

Confirming task execution

Remember our example with skipping migrations in the *if statements and loops* chapter?

```
@story('deploy')
    down
    git
    composer
    @if(!$skip_migrate)
        migrate
    @endif
    cache
    routes
    up
@endstory
```

Now, if you ask me, this looks kind of messy. There is a different way to do this.

We just need to tweak our migrate task a bit:

```
@task('migrate', ['confirm' => true])
    cd {{ $path }}
    php artisan migrate --force
@endtask
```

And then, cleanup the story:

```
@story('deploy')
    down
    git
    composer
    migrate
    cache
    routes
    up
@endstory
```

Now if you run `envoy run deploy`, you will be prompted to confirm running the migrate task:

```
$envoy run deploy
Are you sure you want to run the [migrate] task? [y/N]:
```

Cleanup

If you read the documentation on Laravel Envoy very closely, you will notice that there is a `@finished` directive.

This directive enables you to execute some PHP code after all the tasks have finished.

```
@finished
    echo "All completed";
@endfinished
```

Notifications (slack)

Out of the box, Envoy supports sending notifications to [Slack](https://slack.com/)³¹⁶ after each task is executed.

To be able to configure this you will need a Slack webhook URL. You need to create an “Incoming WebHooks” custom integration in your Slack control panel. Copy the entire Webhook URL into the `@slack` directive:

³¹⁶<https://slack.com/>

```
@finished
    @slack('webhook-url', '#channel')
@endfinished
```

This completes this tutorial. I think that now I have covered every single feature of Envoy.

Deploying a Laravel App from GitHub to Heroku

Deploy a Laravel application from GitHub to Heroku using Codeship for continuous integration and deployment.

Published at: 31. March, 2017.

In this tutorial we will be setting up continuous integration and deployment from [GitHub](#)³¹⁷ to [Heroku](#)³¹⁸ for a [Laravel](#)³¹⁹ application. First I'll show you how to set up continuous integration on [Codeship](#)³²⁰, and then I'll show you how to deploy the application to from Codeship to Heroku.

I've created a Laravel application called [webflix](#)³²¹ on GitHub that I will be using for this tutorial. You can fork the repository and follow along or you can use your own application.

The main *limitation of Heroku* is that it does not support MySQL databases. You can only opt for a PostgreSQL database. You can always use a remote MySQL database; that will work, but for this tutorial, I have decided on using PostgreSQL database since it simplifies things a lot. Laravel supports PostgreSQL database out of the box.

If you already have a working application that uses a MySQL database, you can follow [this article](#)³²² to migrate your MySQL database to a PostgreSQL database.

About Webflix

[Webflix](#)³²³ is dummy Laravel application which simulates Netflix. You can log in, browse movies and mark movies watched. It uses [Bootstrap 4](#)³²⁴ for the design. Bootstrap 4 is still in the alpha stage, but I found it suitable for this application.

This is how the application looks like once you log in:

³¹⁷<https://github.com/>

³¹⁸<https://www.heroku.com/>

³¹⁹<https://laravel.com/>

³²⁰<https://codeship.com/>

³²¹<https://github.com/laravelista/webflix>

³²²<https://devcenter.heroku.com/articles/heroku-mysql>

³²³<http://webflix-laravelista.herokuapp.com/>

³²⁴<https://v4-alpha.getbootstrap.com/>

Deploying a Laravel App from GitHub to Heroku

327

Webflix Home John Doe Log out

The LEGO Batman Movie Bruce Wayne must not only deal with the criminals of Gotham City, but also the responsibility of raising a boy he adopted. Duration: 1h 44min Release date: 10.02.2017 Mark as watched	Fifty Shades Darker While Christian wrestles with his inner demons, Anastasia must confront the anger and envy of the women who came before her. Duration: 1h 58min Release date: 10.02.2017 Mark as watched	John Wick: Chapter 2 After returning to the criminal underworld to repay a debt, John Wick discovers that a large bounty has been put on his life. Duration: 2h 02min Release date: 10.02.2017 Mark as watched	Fist Fight When one school teacher gets the other fired, he is challenged to an after-school fight. Duration: 1h 31min Release date: 17.02.2017 Mark as watched
Split Three girls are kidnapped by a man with a diagnosed 23 distinct personalities, they must try to escape before the apparent emergence of a frightful new 24th. Duration: 1h 57min Release date: 20.01.2017 Mark as watched	The Great Wall European mercenaries searching for black powder become embroiled in the defense of the Great Wall of China against a horde of monstrous creatures. Duration: 1h 43min Release date: 17.02.2017 Mark as watched	Collide An American backpacker gets involved with a ring of drug smugglers as their driver, though he winds up on the run from his employers across Cologne high-speed Autobahn. Duration: 1h 39min Release date: 24.02.2017 Mark as watched	As You Are Set in the early 1990's, "As You Are" is the telling and retelling of a relationship between three teenagers as it traces the course of their friendship through a construction of disparate memories prompted by a police investigation. Duration: 1h 50min Release date: 24.02.2017 Mark as watched

Webflix home page

To be able to mark movies watched, you first have to log in:

Webflix Home Log in

Login
 E-Mail Address

 Password

☐ Remember Me

Webflix login page

User credentials

Email: test@test.com Password: secret

Tests

Webflick comes with two functional tests written with [Codeception](http://codeception.com/)³²⁵. First test tests if the user can log in. The second test tests if the user can mark the movie as watched.

To run the tests execute `vendor/bin/codecept run functional`.

Codeception PHP Testing Framework v2.2.9

Powered by PHPUnit 5.7.14 by Sebastian Bergmann and contributors.

```
Functional Tests (2) -----\
-----\
-----\
â€” LoginCept: Login to my account (1.80s)
â€” WatchMovieCept: Mark a movie as watched (0.25s)
-----\
-----\
-----\
```

Time: 5.2 seconds, Memory: 22.00MB

OK (2 tests, 5 assertions)

Tests are located in `tests/functional` directory.

The purpose of this application is to demonstrate how a working Laravel application, that is hosted on [GitHub](https://github.com/)³²⁶, can be easily added to Codeship for continuous integration, and deployment to [Heroku](https://www.heroku.com/)³²⁷.

Codeship

We now have a Laravel application repository hosted on GitHub. Now, I will show you how to connect your repository to Codeship and run our first integration. Later, I will show you how to use Codeship to automatically deploy our application to Heroku.

If you haven't already, be sure to create a free account on [Codeship](https://codeship.com/)³²⁸. You can sign up with your Github, GitLab or Bitbucket account, or you can use your own email.

³²⁵ <http://codeception.com/>

³²⁶ <https://github.com/>

³²⁷ <https://www.heroku.com/>

³²⁸ <https://codeship.com/>

Free plan & pricing

There are two Codeship platforms: Basic and Pro.

Basic

“Codeship Basic: A simple out-of-the-box Continuous Integration service that just works.” - [source](#)³²⁹

In Basic, there is a Free, Starter and Essential Plan. Each plan offers 100 builds per month, 1 concurrent build, 1 parallel test pipeline, unlimited projects and unlimited users.

In Basic, you get a “shared” virtual machine preinstalled with almost everything you need. You can choose some things like PHP versions etc. It is something like [Laravel Homestead](#)³³⁰. You get a virtual machine with preinstalled software that just works. Then, it is up to you to install your application and run tests on it.

Free plan is free :) as the name says. You get to use it with unlimited open source and private projects. This is what we will be using for this tutorial.

Pro

“Codeship Pro: A fully customizable Continuous Integration service with native Docker support.” - [source](#)³³¹

Pro is pro. You get a dedicated virtual machine which you can configure however you like to install and test your application. If you are using [Docker](#)³³² to set up your development environment, you can use it in Codeship Pro also.

Pricing is a bit higher than for Basic, but that is to be expected. You can read more about Codeship Pro [features](#)³³³.

Let's get started!

Continuous Integration

Log in to your Codeship account. Once logged in, you will be asked to connect your first project to Codeship:

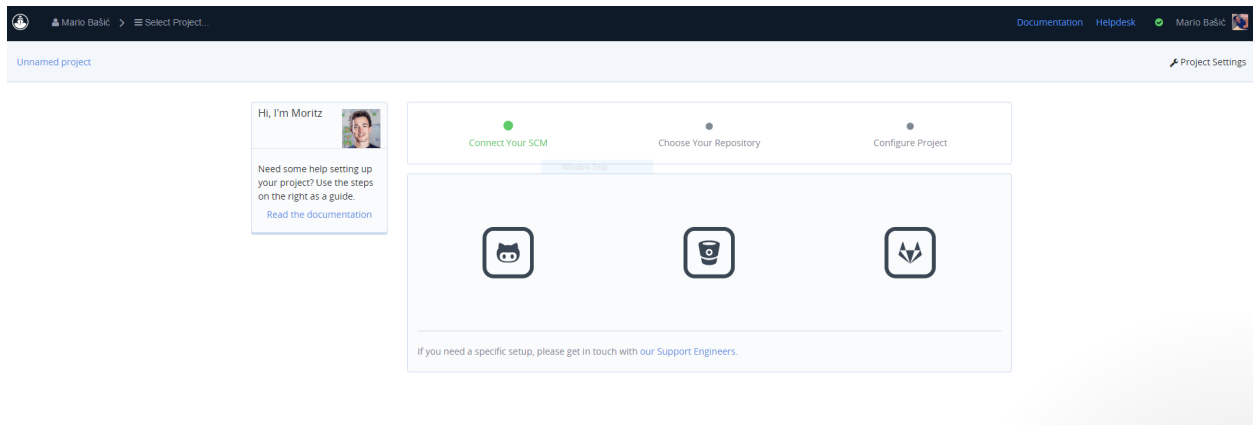
³²⁹<https://codeship.com/pricing/basic>

³³⁰<https://laravel.com/docs/5.4/homestead>

³³¹<https://codeship.com/pricing/pro>

³³²<https://www.docker.com/>

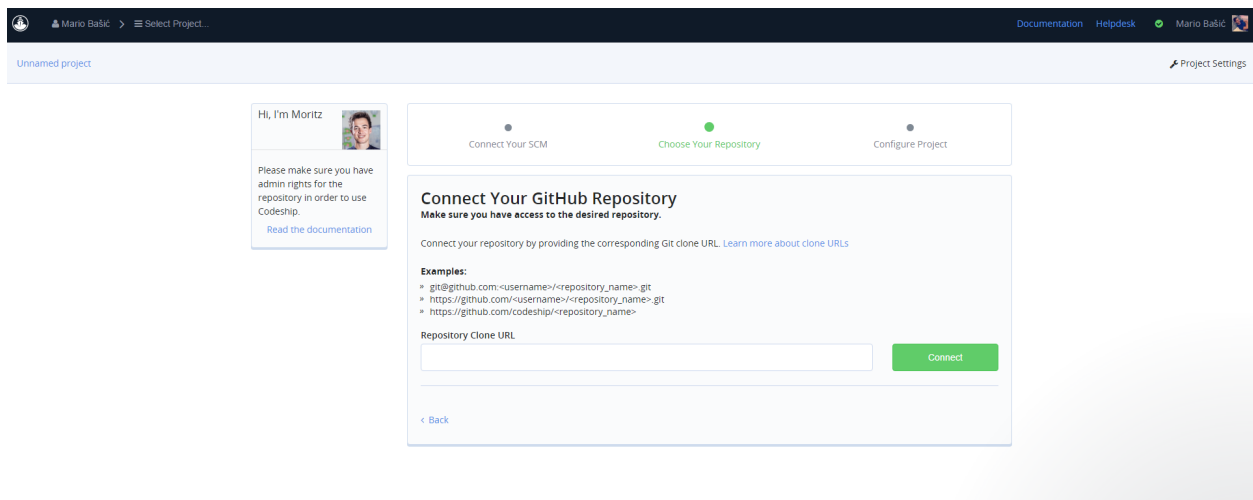
³³³<https://codeship.com/features/pro>



Connect Your SCM

We will choose the GitHub logo here.

On the next screen it will ask you to connect your GitHub repository:



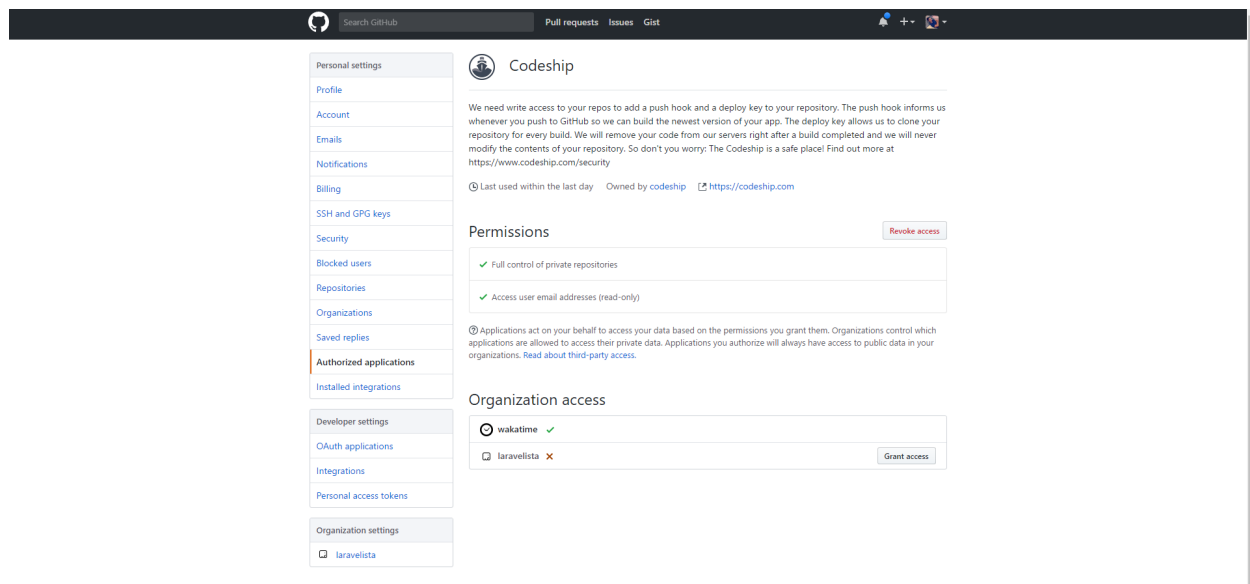
Choose Your Repository

Here, paste the URL of your repository. I will use the repository URL for Webflir: `https://github.com/laravelista`

If you get *“Your organization restricts third-party applications access, please make sure Codeship has access.”*, when you press *Connect*, you have to head over to [Codeship application on GitHub](https://github.com/settings/connections/applications/457423eb34859f8eb490)³³⁴ and in the section labeled **Organization access** either click on:

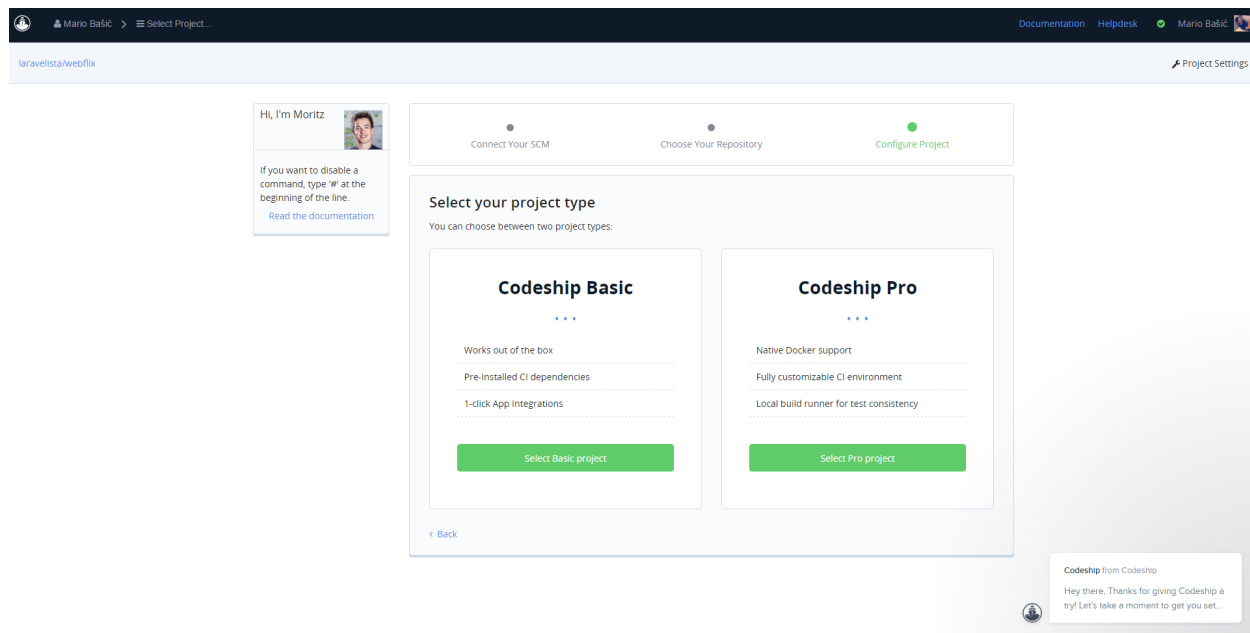
- *Request access* if you are not an administrator for the organization or
- *Grant access* if you are an administrator

³³⁴ <https://github.com/settings/connections/applications/457423eb34859f8eb490>



Codeship application on GitHub

After you have completed the step above, paste the repository URL in the textbox and press *Connect*. You will now be asked to select your project type: Basic or Pro.



Basic or Pro

Click on *Select Basic project*, since that is what we will use for this tutorial.

Next, it will ask you to configure *Setup Commands* and *Test Commands*.

In *Setup Commands* choose *I want to create my own custom commands*, and enter bellow:

```
phpenv local 7.1
composer install --prefer-source --no-interaction
```

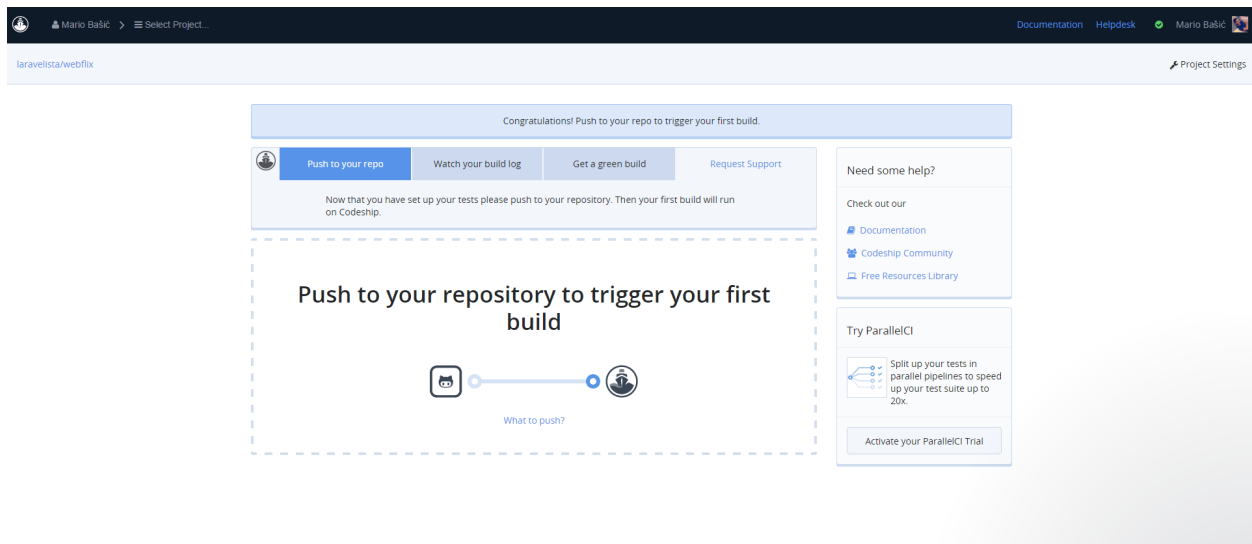
The first command tells Codeship to change the version of PHP to 7.1. The default version used is 5.5. Once the machine is created, the repository of your application will be copied onto it. The next command will use Composer to install the required dependencies for our application.

Since we are using Codeception, in *Test Commands* you only need to enter:

```
vendor/bin/codecept run functional
```

Next, click on *Save and go to dashboard*.

You will be instructed to push to your repository to trigger your first build, but wait.



To trigger or not to trigger

There are a few things that we first need to configure, like our environment variables. On the upper right parts of the page, you will see *Project Settings*. Click on that, and then on *Environment Variables*.

PostgreSQL

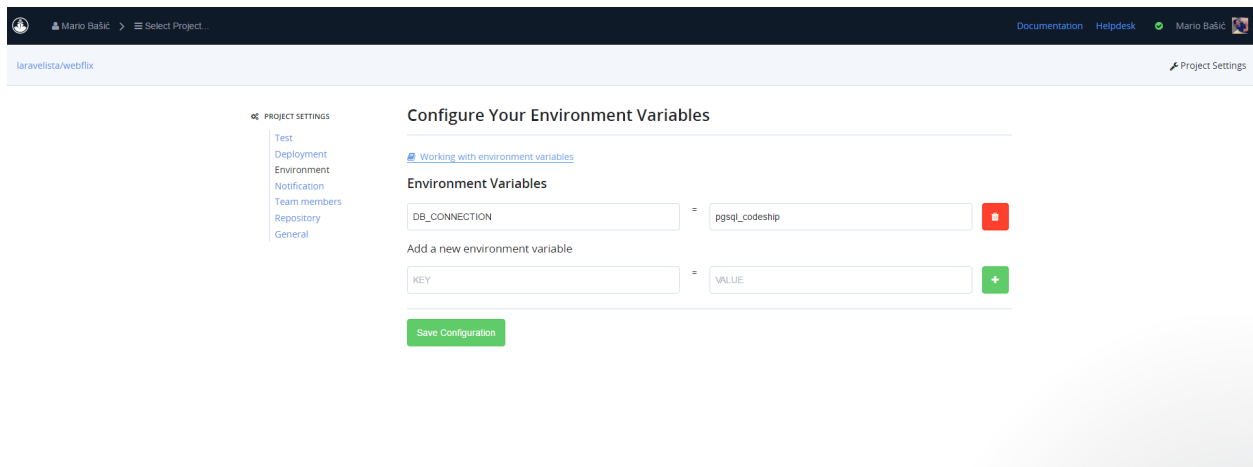
Since our application uses PostgreSQL, we will need to create a new database connection in our Laravel application for Codeship. This connection will only be used for testing on Codeship.

In your application open file `config/database.php` and under connections add this:

```
'pgsql_codeship' => [
  'driver' => 'pgsql',
  'host' => 'localhost',
  'port' => '5432',
  'database' => 'development',
  'username' => env('PGUSER'),
  'password' => env('PGPASSWORD'),
  'charset' => 'utf8',
  'prefix' => '',
  'schema' => 'public',
  'sslmode' => 'prefer',
],
```

You can view the commit [here](#)³³⁵.

Back to *Environment variables*. We need to set an environment variable to tell our application to use our newly created database connection. Under *Key* enter DB_CONNECTION and in *Value* enter pgsql_codeship.



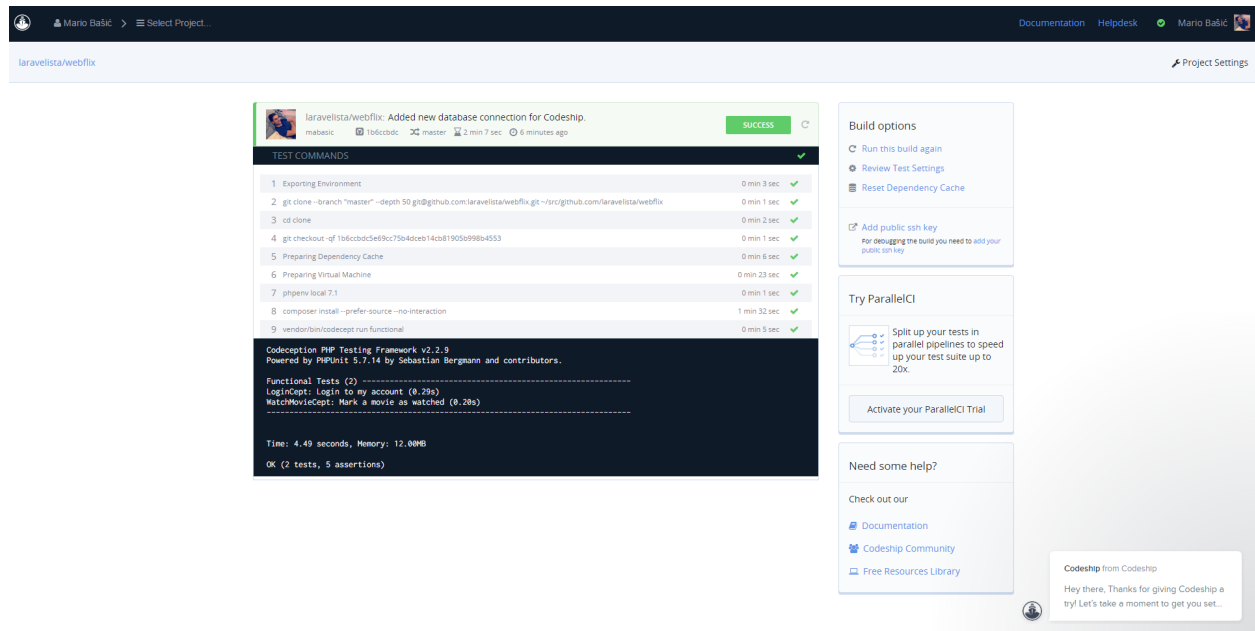
Environment Variables

Now click on *Save Configuration*.

Let's get our first green build.

Make a change in your application, commit the change and push to GitHub. You should see a new build on the Codeship dashboard page saying that the build is *running*. When the build completes, you should see a green *Success* message on that build.

³³⁵ <https://github.com/laravelista/webflix/commit/1b6ccbd5e69cc75b4dceb14cb81905b998b4553>



The screenshot shows a CodeShip build interface for a project named 'laravelista/webfix'. The build status is 'SUCCESS'. The 'TEST COMMANDS' section lists the following steps:

Step	Command	Duration	Status
1	Exporting Environment	0 min 3 sec	✓
2	git clone --branch "master" --depth 50 git@github.com:laravelista/webfix.git --no-github.com:laravelista/webfix	0 min 1 sec	✓
3	cd clone	0 min 2 sec	✓
4	git checkout -f 166c0d5e69c73b4d0eb14c81905d99b04553	0 min 1 sec	✓
5	Preparing Dependency Cache	0 min 6 sec	✓
6	Preparing Virtual Machine	0 min 23 sec	✓
7	phpenv local 7.1	0 min 1 sec	✓
8	composer install --prefer-source --no-interaction	1 min 32 sec	✓
9	vendor/bin/codecept run functional	0 min 5 sec	✓

The terminal output shows the Codeception framework running functional tests successfully:

```
Codeception PHP Testing Framework v2.2.9
Powered by PHPUnit 5.7.14 by Sebastian Bergmann and contributors.

Functional Tests (2) -----
LoginCapt: Login to my account (0.29s)
WatchMovieCapt: Mark a movie as watched (0.20s)
-----
Time: 4.49 seconds, Memory: 12.00MB
OK (2 tests, 5 assertions)
```

On the right side of the interface, there are sections for 'Build options' (Run this build again, Review Test Settings, Reset Dependency Cache), 'Try ParallelCI' (Split up your tests in parallel pipelines to speed up your test suite up to 20x, Activate your ParallelCI Trial), and 'Need some help?' (Check out our Documentation, CodeShip Community, Free Resources Library).

Success

If the build fails saying *Failed*, recheck the steps in this tutorial and/or click on that build to see the details and locate in which step the build failed. You can even remotely log into the virtual machine using SSH to inspect the application.

From this point on, every test commit you make will trigger a build and you will get a notification if the build fails.

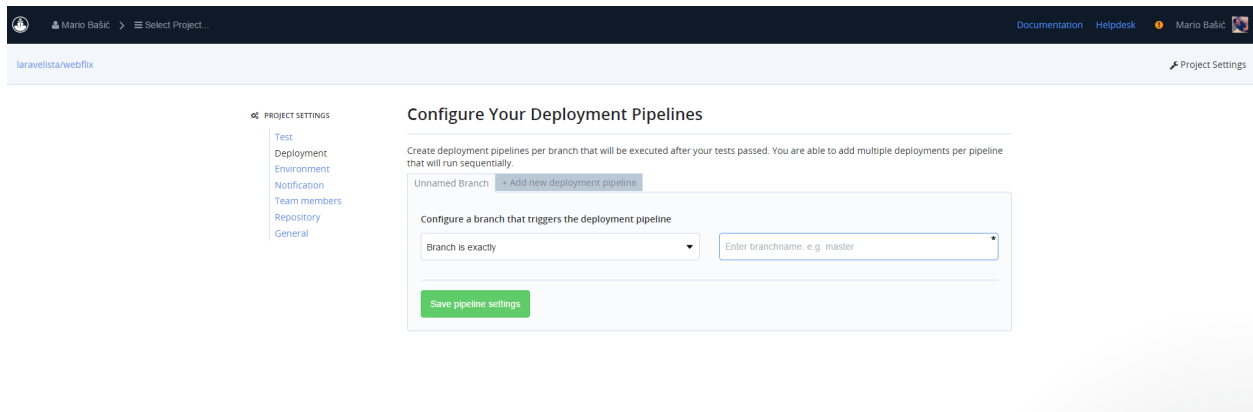
Continuous integration complete!

Continuous Deployment

We will now set up automatic deployment to [Heroku](https://www.heroku.com/)³³⁶ on each successful build on the master branch.

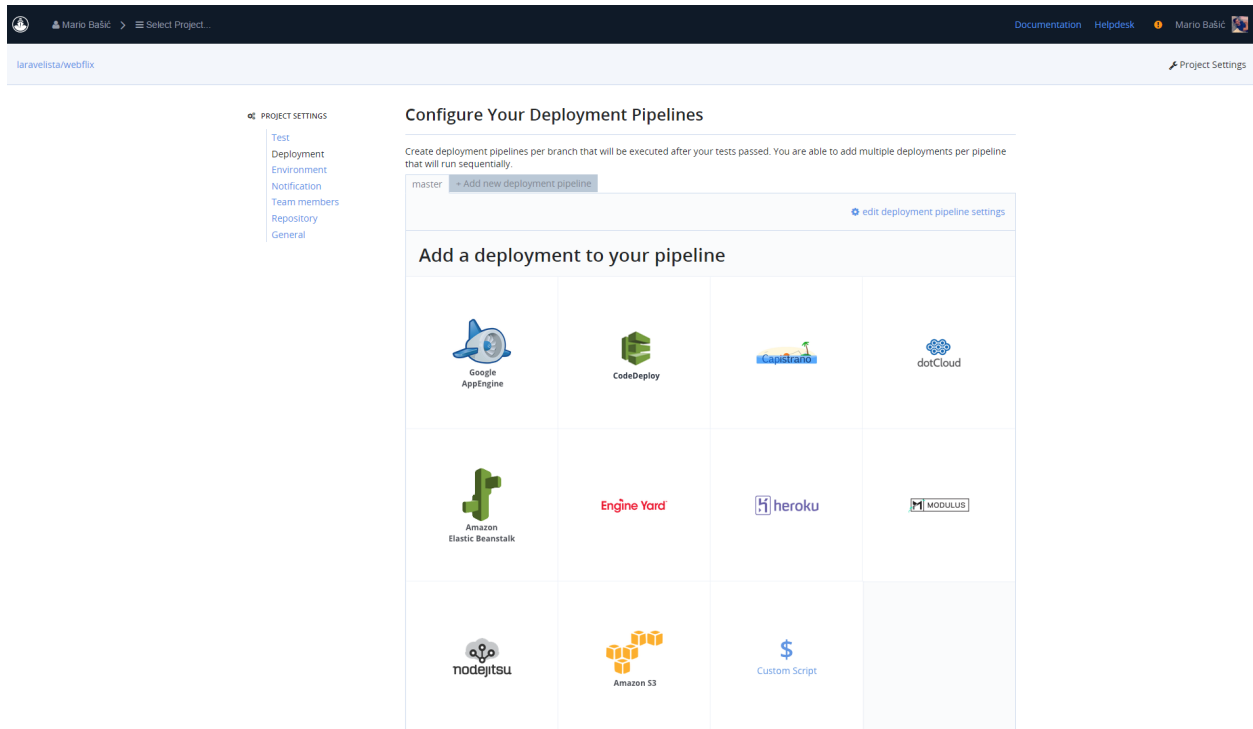
On the upper-right part of the page, you will see *Project Settings*. Click on that, and then click on *Deployment*.

³³⁶<https://www.heroku.com/>



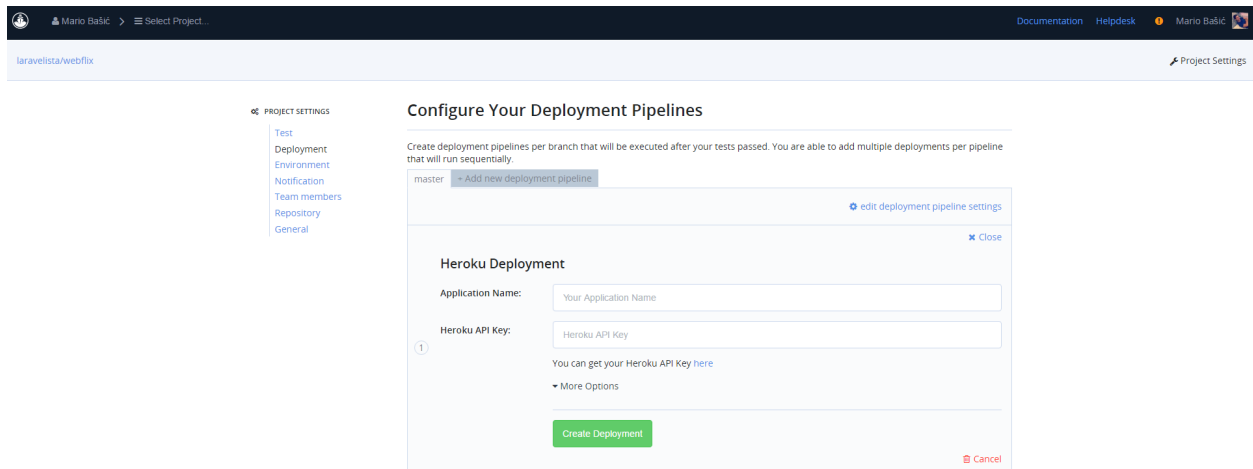
Deployment

Make sure that it says “Branch is exactly”, and then in the branch name enter `master`. Click on *Save pipeline settings*. You will be presented with a grid of supported deployment ways.



Deployments grid

Click on the Heroku logo. You will be greeted with the following screen:



Heroku deployment

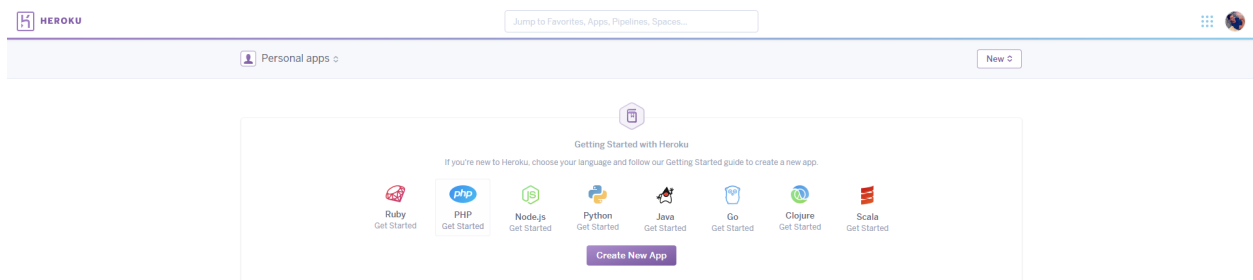
Pause here.

Now, the next part we have to do on Heroku. We will create an account, create an application and obtain the API key.

Heroku

If you don't have an account on Heroku, you will have to sign up. You can do that from the [Heroku home page](https://www.heroku.com/)³³⁷ by clicking on the *Sign up for free* button.

Once you log in, you will be on the [dashboard page](https://dashboard.heroku.com/apps)³³⁸.

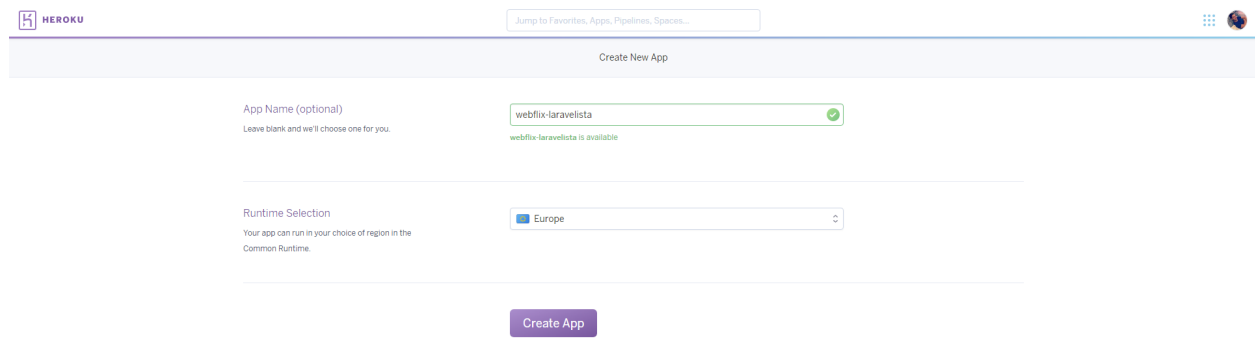


Heroku dashboard

From here you can create a new app by clicking on *Create New App* button. On the following screen, you will be asked to enter the application name or one will be chosen for you. You can also select the region of your application: Europe or United States.

³³⁷ <https://www.heroku.com/>

³³⁸ <https://dashboard.heroku.com/apps>



The screenshot shows the Heroku 'Create New App' form. At the top, there is a search bar with the text 'Jump to Favorites, Apps, Pipelines, Spaces...'. Below this is a section titled 'Create New App'. The first section is 'App Name (optional)' with the instruction 'Leave blank and we'll choose one for you.' The input field contains 'webflix-laravelista' and has a green checkmark icon to its right. Below the input field, it says 'webflix-laravelista is available'. The second section is 'Runtime Selection' with the instruction 'Your app can run in your choice of region in the Common Runtime.' The dropdown menu is set to 'Europe'. At the bottom of the form is a purple button labeled 'Create App'.

Heroku dashboard

I have entered webflix-laravelista and selected Europe.

There are three things that we need to obtain from Heroku:

- Application name
- API key
- Application URL

So, in my case, the application name is webflix-laravelista. Application URL can be found if you navigate to settings and then under *Domains* you will see the URL. In my case, it is <https://webflix-laravelista.herokuapp.com/>.

Deploying a Laravel App from GitHub to Heroku

338

The screenshot shows the Heroku dashboard for the application 'webfix-laravelista'. The top navigation bar includes the Heroku logo, a search bar, and links for 'Personal apps' and 'webfix-laravelista'. Below the navigation bar, there are tabs for 'Overview', 'Resources', 'Deploy', 'Metrics', 'Activity', 'Access', and 'Settings'. The 'Settings' tab is currently selected.

The 'Settings' page is divided into several sections:

- Name:** webfix-laravelista
- Config Variables:** A section with a 'Reveal Config Vars' button. Below it, a note states: 'Config vars change the way your app behaves. In addition to creating your own, some add-ons come with their own.'
- Info:** A table showing application details:

Region	Europe
Stack	cedar-14
Framework	No framework detected
Slug size	No slug detected
Heroku Git URL	https://git.heroku.com/webfix-laravelista.git
- Buildpacks:** A section with an 'Add buildpack' button. Below it, a note states: 'Buildpacks are scripts that are run when your app is deployed. They are used to install dependencies for your app and configure your environment. Find New Buildpacks at Heroku Elements.'
- Domains and certificates:** A section with a 'Domain' field showing 'Your app can be found at https://webfix-laravelista.herokuapp.com/'. Below it, an 'SSL' field shows 'Upgrade to paid dynos to configure Heroku SSL'. There is also a 'Custom Domains' section with an 'Add domain' button and a note: 'Custom domains will appear here. Custom domains allow you to access your app via one or more non-Heroku domain names (for example, www.yourcustomdomain.com).'
- Transfer Ownership:** A section with a 'Select a new owner...' dropdown and a 'Transfer' button. Below it, a note states: 'Transfer this app to your personal account or to an organization or team of which you are a member. Learn More.'
- Maintenance Mode:** A section with a toggle switch labeled 'Maintenance mode is off' and a 'More info' link.
- Delete App:** A section with a 'Delete app...' button and a note: 'Deleting your app and its add-ons is irreversible.'

The footer of the page includes links for 'heroku.com', 'Blogs', 'Careers', 'Documentation', 'Support', 'Terms of Service', 'Privacy', 'Cookies', and '© 2017 Salesforce.com'.

Application name

To obtain the API key, you have to go to *Account Settings*. Click on your profile image and then on *Account Settings*. At the bottom of the page you will see a section called *API Key*.

HEROKU

Jump to Favorites, Apps, Pipelines, Spaces...

Two-factor Authentication
Enable to give your Heroku account an extra layer of security.
[Set Up Two-factor Authentication...](#)

Privacy
☒ Allow use of third-party analytics services.
For more information, view our [privacy policy](#).

SSH Keys
Need help? Checkout out our guide to [generating SSH keys](#).
There are no SSH keys yet.
You can register new SSH keys to enable command line access to your apps.
[Add New SSH Key](#)

API Key
.....
[Reveal](#)
[Regenerate API Key...](#)

Close Account
You must delete all apps on this account first.
Warning: Closing your account is irreversible.
[Close this account...](#)

Application name

Click on *Reveal* to see your API key.

We now have everything needed to continue setting up deployment on Codeship.

Navigate back to our deployment page on Codeship and enter the obtained information. (Click on *More options* to see the application URL input box.)

Once done, click on the *Create Deployment* button.

Before we continue there are a few things that we have to do before Laravel and Heroku are compatible.

On Heroku, go to *Application settings*, and in the section *Config Variables* click on *Reveal Config Vars*. Here enter these key/value pairs:

- APP_KEY: base64:Th9tL32tWxfpvCBJ00FiAaJqb81v+toYKsZ29A6L7Y0=
- APP_LOG: errorlog
- DB_CONNECTION: pgsql_heroku

Now, to install PostgreSQL add-on, go to *Application overview* page and click on *Configure Add-ons*.

Personal apps > webflick-laravelista

Overview Resources Deploy Metrics Activity Access Settings

Installed add-ons \$0.00/month Configure Add-ons

There are no add-ons for this app
You can add add-ons to this app and they will show here. [Learn more](#)

Dyno formation \$0.00/month Configure Dynos

This app is using free dynos

web	vendor/bin/heroku-php-nginx public/	ON

Collaborator activity Manage Access

mario@laravelista.com	4 deploys
-----------------------	-----------

Add-ons

On the next screen in the search box, search for Heroku Postgres and click on it. A modal window will pop up prompting you to choose a plan. Select *Hobby Dev - Free* and click on *Provision* button. You now have a database connected to your Heroku application.

Now we have to tell Heroku to configure Nginx to serve our application from the `/public` directory and to use a special Nginx config file. In your Laravel application, first, you have to add a Procfile file in your repository root. The Procfile should contain this code:

```
web: vendor/bin/heroku-php-nginx -C nginx_app.conf public/
```

Then, we now need to instruct Nginx to handle Laravel by removing the `index.php` part of the URI. Create a file called `nginx_app.conf` in your repository root, and inside it paste this code:

```
location / {
    try_files $uri @rewriteapp;
}

location @rewriteapp {
    rewrite ^(.*)$ /index.php$1 last;
}
```

Also, we need to create a database connection for `pgsql_heroku`. Go to your `config/database.php` and at the top of the file, just below `<?php` and above `return` [add this code:

```
$host = null;
$username = null;
$password = null;
$databse = null;

if(!is_null(env('DATABASE_URL')))
{
    $url = parse_url(env("DATABASE_URL"));

    $host = $url["host"];
    $username = $url["user"];
    $password = $url["pass"];
    $databse = substr($url["path"], 1);
}
```

Then, under connections add this code:

```
'pgsql_heroku' => [
    'driver' => 'pgsql',
    'host' => $host,
    'port' => 5432,
    'database' => $databse,
    'username' => $username,
    'password' => $password,
    'charset' => 'utf8',
    'prefix' => '',
    'schema' => 'public',
    'sslmode' => 'prefer',
],
```

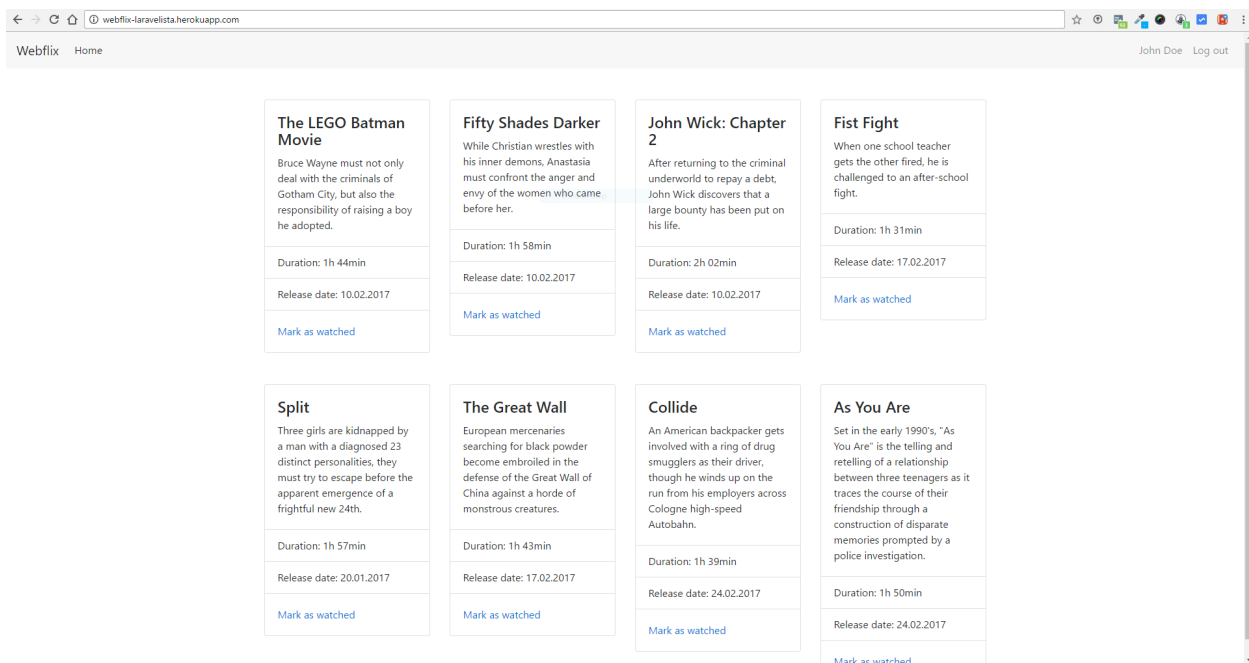
One last step, on each deployment to Heroku we should automatically run migrations on our database. To do so, open `composer.json` file and append this line under `post-install-cmd`:

```
"php artisan migrate --force"
```

Now your post-install-cmd script looks like this:

```
"post-install-cmd": [
    "Illuminate\\Foundation\\ComposerScripts::postInstall",
    "php artisan optimize",
    "php artisan migrate --force"
],
```

Great, now make a commit and push. You should see a new build appear in Codeship. If the build turns green (success; and it should at this point) you can browse to your application URL and see that it works. In my case, it is <http://webflix-laravelista.herokuapp.com/>³³⁹.



Application online

This completes this tutorial. You have now implemented continuous integration and continuous deployment from GitHub to Heroku on a Laravel application using Codeship. **Thank you for reading.**

In my opinion, I think that this is the most detailed and comprehensive tutorial on how to deploy Laravel to Heroku.

³³⁹ <http://webflix-laravelista.herokuapp.com/>

If you have any questions, leave me a comment below and I will respond promptly.

Also, if you need more resources on the subject, take a look at the sources below. They helped me a lot during this tutorial. Have a great week!

Sources

- [Laravel Database: Getting Started](#)³⁴⁰
- [Migrating from MySQL to Postgres on Heroku](#)³⁴¹
- [Bootstrap 4](#)³⁴²
- [GitHub](#)³⁴³
- [Heroku](#)³⁴⁴
- [Codeception](#)³⁴⁵
- [Laravel](#)³⁴⁶
- [Laravel on Heroku - Using a PostgreSQL database](#)³⁴⁷
- [Installing a Laravel app on Heroku](#)³⁴⁸
- [Getting Started with Laravel on Heroku](#)³⁴⁹
- [Customizing Web Server and Runtime Settings for PHP](#)³⁵⁰
- [Stack overflow - Laravel nginx.conf with official Heroku php buildpack?](#)³⁵¹
- [Getting started with Laravel and PHP on Heroku](#)³⁵²

³⁴⁰<https://laravel.com/docs/5.4/database>

³⁴¹<https://devcenter.heroku.com/articles/heroku-mysql>

³⁴²<https://v4-alpha.getbootstrap.com/>

³⁴³<https://github.com/>

³⁴⁴<https://www.heroku.com/>

³⁴⁵<http://codeception.com/>

³⁴⁶<https://laravel.com/>

³⁴⁷<https://mattstauffer.co/blog/laravel-on-heroku-using-a-postgresql-database>

³⁴⁸<https://mattstauffer.co/blog/installing-a-laravel-app-on-heroku>

³⁴⁹<https://devcenter.heroku.com/articles/getting-started-with-laravel>

³⁵⁰<https://devcenter.heroku.com/articles/custom-php-settings>

³⁵¹<http://stackoverflow.com/questions/28956500/laravel-nginx-conf-with-official-heroku-php-buildpack#28973459>

³⁵²<http://www.ryan kent.ca/getting-started-with-laravel-and-php-on-heroku/#comment-1387220063>

Server Administration

For the brave, there are a couple of tutorials on installing Nginx, PHP7, obtaining certificates with Let's encrypt and bash scripting.

Bash Scripting Introduction

This is an introductory tutorial to Bash scripting and the purpose of it is to encourage you to be more productive while using a shell.

Published at: 26. May, 2016.

I have been waiting a long time to write a tutorial like this one. To me learning Bash scripting is like learning a brand new programming language where the possibilities are endless and the learning never stops.

Why Bash scripting? The most simple answer to this question is this. *You can use `..` instead of `cd`* ... If you are anything like me, this will surely get you interested.

In this tutorial, I will show you how to use variables, aliases, and functions with examples. All examples provided here are the one I personally use. It does not matter on which OS you are, Bash is the same everywhere.

To install on Windows, see here: [Git Bash](#).

Many of you may wonder how does this tutorial relate to Laravel. The answer is simple. To access Homestead globally from anywhere on your machine, you use a Bash function mentioned in the Laravel documentation [Accessing Homestead Globally](#)³⁵³. I don't mean to brag *<i class="fa fa-smile-o">*, but I wrote the function mentioned there:

```
function homestead() {
    ( cd ~/Homestead && vagrant $* )
}
```

Variables

This is a great starting point for you. Before we do any actual work, you need to locate your `.bash_profile` file. Usually, it can be found in your home directory. That would be `~/.bash_profile`. On Windows that is `C:\Users\<your-user-profile-name>\.bash_profile`. If you can't find the file there, create a new blank file called `.bash_profile`.

All code provided here is to be written in that file. May the games begin.

Declaring a variable is simple:

³⁵³<https://laravel.com/docs/5.2/homestead#accessing-homestead-globally>

```
REPOSITORIES_ROOT="/c/Repositories"
HOMESTEAD_DIR="$REPOSITORIES_ROOT/Homestead"
```

Let me explain. All variables by default are global. The first line declares a variable called `REPOSITORIES_ROOT` and assigns it a value that points to the location on my PC where I hold all the repositories that I work on.

The second line declares a variable called `HOMESTEAD_DIR` and assigns it a value by combining the variable we just created `REPOSITORIES_ROOT` and appending `/Homestead` to it. If we were to echo that variable, we would get `/c/Repositories/Homestead`.

Now you know how to declare variables and use variables in other variables so that you can reuse variables as needed.

Aliases

Variables by itself are not doing pretty much anything. To make use of variables, we use aliases and functions. First I will show you aliases since they are simpler (They can get pretty complex, but at that point, you should consider using functions).

First, let me show you some alias samples:

```
alias ..="cd .."
alias vm="ssh vagrant@127.0.0.1 -p 2222"
alias rr="cd $REPOSITORIES_ROOT"
```

The first alias changes your current working directory to one above. It is the standard way of navigating the file system. `cd` stands for “change directory”, `..` (two dots) tell the function `cd` to navigate one directory up aka go to the parent directory.

The second alias is taken from the Laravel documentation [Connecting via SSH](https://laravel.com/docs/5.0/homestead#daily-usage)³⁵⁴. In my opinion, using `vm` instead of `homestead` `ssh` is much faster. This alias uses `ssh` to connect to the Homestead virtual machine on your host machine. Username is `vagrant`, host is `127.0.0.1` and the port is `2222`.

Last alias is my personal alias. I often find myself in different directories, like navigating in public `js` files or tracking down a file in `node_modules`. For those cases whenever I want to “bail” back to the “repository root” (That is the name I have given to the folder that holds all my repositories/projects) I type `rr` and I’m there... I should probably change it to `bail` or maybe not. It is completely optional and up to you. That is the beauty of Bash scripting.

The aliases, functions, and variables used by a user are completely unique and personal to that user. Someone may prefer this name for a variable or that name for an alias. *Do what you want and*

³⁵⁴<https://laravel.com/docs/5.0/homestead#daily-usage>

express yourself or don't, it is completely optional and up to you. I am here to tell you that you have options.

Back to the tutorial. You need to add the above code to `~/ .bash_profile` file and restart your shell for the changes to take effect.

Functions

Now I will show you two functions and explain what they do. The first one you have seen before, it is for accessing Homestead globally:

```
function homestead() {
    ( cd $HOMESTEAD_DIR && vagrant $* )
}
```

We declare a function called `homestead` and inside create a sub-shell by enclosing the function code in parentheses (`... function code ...`). What is a sub-shell? A sub-shell is a shell that executes in the background without you seeing it. The reason why we want a sub-shell here is because we don't want to change our working directory to where our Homestead directory is, but we still want to do that behind the scenes before we call the `vagrant` function. I hope that it makes sense to you. The code inside the parentheses changes the current working directory to the `HOMESTEAD_DIR` variable we declared before and executes a function `vagrant` to which it passes everything `$*` we entered after `homestead up`, `homestead suspend`, `homestead ssh` etc..

The important things to remember here are:

- Parentheses () invoke a subshell
- We can use variables by prefixing the with \$
- To access all parameters passed to the function we use \$*
- To write two commands in one line we use && or ;
- We declare function by using `function` keyword before the function name

Now I will show you a function that I have created because I am lazy to type *<i class="fa fa-smile-o"></i>*:

```
function goto() {
    ssh ${2:-username}@${1} -p ${3:-666}
}
```

I manage a lot of servers and I find it very tiresome to have to type `ssh someuser@some-server.com -p xyz`, so I have created a function that enables me to type `goto some-server.com` and it already has my default username and port. I still have an option to change the username and port if I require.

No, my default username is not `username` and my default port is not 666. I have typed that here to prevent abuse.

Key things to notice here are:

- To use a specific parameter use `${1}` where you replace 1 with the parameter you want. It goes up to number 9 I think.
- To provide a default value, if the parameter is not provided, use `${1:my-favorite-server.com}`.

There you are. You now know how to write Bash functions. Good job.

Room for improvement

I have barely scratched the surface on Bash scripting in this tutorial. To learn more you should google it. Google is your friend here. My favorite resources when I want something done with Bash are:

- [Writing a Simple Bash Script](https://www.linux.com/learn/writing-simple-bash-script)³⁵⁵ - useful reminder of the basics
- [Bash Guide for Beginners](http://tldp.org/LDP/Bash-Beginners-Guide/html/)³⁵⁶ - It says for beginners, but it really isn't haha
- [Advanced Bash-Scripting Guide](http://tldp.org/LDP/abs/html/)³⁵⁷ - almost the same as above, but with more of everything. Very useful.

³⁵⁵ <https://www.linux.com/learn/writing-simple-bash-script>

³⁵⁶ <http://tldp.org/LDP/Bash-Beginners-Guide/html/>

³⁵⁷ <http://tldp.org/LDP/abs/html/>

Upgrade to PHP 7 on Ubuntu 14.04

If you are anything like me, then you have a bunch of servers still running PHP 5.6. The time has come for you and me to upgrade to PHP 7.

Published at: 11. July, 2016.



View Source Code

Source code for this tutorial is available [here](#)³⁵⁸.

I have been finally forced to upgrade to PHP 7 by a single Laravel package. This is very sad but ok. The good thing is that you can serve sites with both PHP 5.6 and PHP 7 at the same time, so in that way, you can still run old applications that require PHP 5.6 and run new applications that run on PHP 7.

I manage a bunch of servers and I'm planning on upgrading them all, so I will write down a small and simple [gist](#)³⁵⁹ on how to do this. The site that I will be upgrading to PHP 7 in this tutorial is this site [Laravelista](#)³⁶⁰. There are a few other sites on the same server, but I will leave those on PHP 5.6 until needed.

Upgrading for the sake of upgrading is a waste of time if you ask me. I like to have everything on the latest release, but sometimes time spent does not convert to money earned.

Prerequisites

Since this is a tutorial on how to upgrade, I won't be dealing with installing and configuring the server for serving Laravel applications. I assume that you already have done that part.

This tutorial assumes that you are running PHP 5.x on Ubuntu 14.04 operating system and using PHP-FPM with Nginx. It also assumes that you have a non-root user configured with sudo privileges for administrative tasks.

³⁵⁸ <https://gist.github.com/mabasic/b9f4cc5706bc34a5dff4bd930585cd98>

³⁵⁹ <https://gist.github.com/mabasic/b9f4cc5706bc34a5dff4bd930585cd98>

³⁶⁰ <https://laravelista.com>

Adding a PPA for PHP

PPA stands for Personal Package Archive and is an Apt repository hosted on [Launchpad](https://launchpad.net/)³⁶¹. Ondrej SurÃ½ maintains the PHP packages for Debian.

Add the PPA to your system's Apt sources:

```
sudo add-apt-repository ppa:ondrej/php
```

You will be asked for your sudo password and be presented with a description of the PPA. Press Enter to continue.

Once installed, you have to update your local packages cache to include the content of the newly added PPA:

```
sudo apt-get update
```

Installing PHP 7

As mentioned before, this tutorial assumes that you are running Nginx with PHP-FPM, so we will install PHP 7, PHP-FPM 7 and every other PHP dependency that Laravel requires with the following command:

```
sudo apt-get install php7.0 php7.0-fpm php7.0-mysql php7.0-curl php7.0-mcrypt php7.0-mbstring php7.0-gd php7.0-zip php7.0-xml
```

After the installation has completed, you have to enable the mods:

```
sudo phpenmod mcrypt
sudo phpenmod mbstring
```

You can verify that you are now using PHP 7 in your command line by typing `php -v`. You should get an output similar to this one:

³⁶¹<https://launchpad.net/>

```
PHP 7.0.8-3+deb.sury.org~trusty+1 (cli) ( NTS )
Copyright (c) 1997-2016 The PHP Group
Zend Engine v3.0.0, Copyright (c) 1998-2016 Zend Technologies
    with Zend OPcache v7.0.8-3+deb.sury.org~trusty+1, Copyright (c) 1999-2016, b\
y Zend Technologies
```

Updating Nginx sites to use new socket path

In order to tell Nginx which sites to run with PHP 7, we need to update the site configuration file and change the UNIX socket it uses from PHP 5.x to PHP 7 path.

Your site configuration is probably located in `/etc/nginx/sites-available` directory. Open your site configuration file and locate the line that says:

```
fastcgi_pass unix:/var/run/php5-fpm.sock;
```

Change that line to:

```
fastcgi_pass unix:/var/run/php/php7.0-fpm.sock;
```

You need to change that line in every site configuration file which you want to run with PHP 7.

Save the changes and restart Nginx with:

```
sudo service nginx restart
```

If everything went smoothly, you should check your website now and you will see that it probably loads much faster than before. At least it seems to me that it does.

This concludes this tutorial.

Secure Nginx with Let's Encrypt on Ubuntu 14.04

Let's Encrypt enables you to obtain and install free TLS/SSL certificates on your web server by using a software client that automates most of the steps required.

Published at: 11. July, 2016.



View Source Code

Source code for this tutorial is available [here](#)³⁶².

I know that I am a little late to the party, but since I had a valid SSL certificate at the time I did not pay any attention to this great new service called [Let's Encrypt](#)³⁶³.

Recently I started working on a new and exciting project that required an SSL certificate. I remembered hearing about something that offered free SSL certificates and decided to give it a chance. The installation and configuration went smoothly without a problem and everything worked.

I am writing this tutorial to show you how to obtain and install a free SSL certificate from Let's Encrypt on Nginx web server running on Ubuntu 14.04. You can view the gist for this tutorial [here](#)³⁶⁴.

Certbot installation

The first step is very simple. We have to install a Let's Encrypt client. There are [many clients](#)³⁶⁵ to choose from, but the officially recommended one is [Certbot](#)³⁶⁶.

To install Certbot, navigate to your user home directory and run commands:

³⁶²<https://gist.github.com/mabasic/81fe616298005818f71f3422a0436bca>

³⁶³<https://letsencrypt.org/>

³⁶⁴<https://gist.github.com/mabasic/81fe616298005818f71f3422a0436bca>

³⁶⁵<https://letsencrypt.org/docs/client-options/>

³⁶⁶<https://certbot.eff.org/>


```
cd ~
```

```
wget https://dl.eff.org/certbot-auto
chmod a+x certbot-auto
```

You will get a response similar to this one:

```
--2016-07-11 05:53:25-- https://dl.eff.org/certbot-auto
Resolving dl.eff.org (dl.eff.org)... 173.239.79.196
Connecting to dl.eff.org (dl.eff.org)|173.239.79.196|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 44115 (43K) [text/plain]
Saving to: 'certbot-auto'

100%[=====\
=====\\
=====>] 44,115      284KB/s   in 0.2s

2016-07-11 05:53:26 (284 KB/s) - 'certbot-auto' saved [44115/44115]
```

Run `./certbot-auto` to let Certbot install its dependencies and update the client code automatically.

If you are getting `InsecurePlatformWarning` during this step, you need to update your Python version to at least 2.7.9.

Obtaining certificates

To obtain a certificate for my website (laravelista.com) I need to enter this command:

```
./certbot-auto certonly --webroot -w /www/laravelista/lux/public -d laravelista.\
com -d www.laravelista.com
```

This command tells Certbot to use a *Webroot plugin* to obtain a certificate for domains: laravelista.com and www.laravelista.com.

In order to obtain certificates for your domains you have to change the domains you want `-d yourdomain.com` and change the webroot location to match your site configuration file in Nginx `-w /www/yoursite/public`.

Once you have entered the command you will be asked for your email address and asked to agree to the terms of service from Let's Encrypt.

Once completed, you will get an important note similar to this one:

IMPORTANT NOTES:

- Congratulations! Your certificate and chain have been saved at `/etc/letsencrypt/live/laravelista.com/fullchain.pem`. Your cert will expire on 2016-10-09. To obtain a new or tweaked version of this certificate in the future, simply run `certbot-auto` again. To non-interactively renew **all** of your certificates, run `"certbot-auto renew"`
- If you lose your account credentials, you can recover through e-mails sent to .
- Your account credentials have been saved in your Certbot configuration directory at `/etc/letsencrypt`. You should make a secure backup of this folder now. This configuration directory will also contain certificates and private keys obtained by Certbot so making regular backups of this folder is ideal.
- If you like Certbot, please consider supporting our work by:

Donating to ISRG / Let's Encrypt: <https://letsencrypt.org/donate>

Donating to EFF: <https://eff.org/donate-le>

Your certificate and private key are now located in `/etc/letsencrypt` directory. You will now have the following files:

- **cert.pem**: Your domain's certificate
- **chain.pem**: The Let's Encrypt chain certificate
- **fullchain.pem**: cert.pem and chain.pem combined
- **privkey.pem**: Your certificate's private key

Configure Nginx to use the new certificate

Now we have to configure our site configuration file to serve the newly obtained certificate.

We do this by opening our site configuration file probably located in `/etc/nginx/sites-available`.

- If you are switching from regular certificate to Let's Encrypt certificate you will probably only have to change the certificate and private key location path.
- If you are installing a certificate for the first time, you will have some extra steps like serving traffic on port 443, adding certificates and configuring protocols and ciphers.

Only updating

If you are only updating, then you have to find `ssl_certificate` and `ssl_certificate_key` keys and change their value to the corresponding value depending on your domain.

```
ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;
```

Save the changes and restart Nginx using `sudo service nginx restart`. If you check your certificate now, you will see that it is only valid for three months. This means that you have successfully installed the certificate.

First time installing

If this is your first time installing a certificate there are some key things to remember.

You need to listen on port 443

```
listen 443;
listen [::]:443; # listen for ipv6
```

You need to include the certificate and private key and enable SSL:

```
ssl on;
ssl_certificate /etc/letsencrypt/live/yourdomain.com/fullchain.pem;
ssl_certificate_key /etc/letsencrypt/live/yourdomain.com/privkey.pem;
```

Configure protocols and ciphers:

#enables all versions of TLS, but not SSLv2 or 3 which are weak and now deprecated.

```
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
```

#Disables all weak ciphers

```
ssl_ciphers "ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:DHE-RSA-AES\
256-GCM-SHA384:DHE-RSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES12\
8-SHA256:ECDHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:DHE-RSA-AES256-SHA256:DHE-RSA\
-AES128-SHA256:DHE-RSA-AES256-SHA:DHE-RSA-AES128-SHA:ECDHE-RSA-DES-CBC3-SHA:EDH-\
RSA-DES-CBC3-SHA:AES256-GCM-SHA384:AES128-GCM-SHA256:AES256-SHA256:AES128-SHA256\
:AES256-SHA:AES128-SHA:DES-CBC3-SHA:HIGH:!aNULL:!eNULL:!EXPORT:!DES:!MD5:!PSK:!R\
C4";
```

```
ssl_prefer_server_ciphers on;
```

This tutorial is not meant to teach you how to configure Nginx to serve HTTPS traffic, you should already know how to do it or google it. If there is a need I will be glad to write a tutorial just for that.

Automating Renewal

As you may have noticed already, the certificate obtained from Let's Encrypt is only valid for 90 days. To set up automatic renewal process we need to setup a Cron entry.

To test that we can renew a certificate we can start a "dry run":

```
cd ~
```

```
./certbot-auto renew --dry-run
```

If that appears to be working correctly we can proceed to add a Cron entry. To open a sudo Cron tab enter:

```
sudo crontab -e
```

At the bottom of the file type:

```
30 2 * * 1 ~/certbot-auto renew --quiet --no-self-upgrade
33 2 * * 1 /etc/init.d/nginx reload
```

This will create a new cron job that will execute every Monday at 2:30h and reload Nginx at 2:33h. The output from the command can be viewed at `/var/log/le-renewal.log`.

Please select a random minute within the hour for your renewal tasks.

This concludes this tutorial.

Using SSH to execute commands on a remote server

Upon a reader request, I will cover installing and using `laravelcollective/remote` package to execute commands on a remote server.

Published at: 13. May, 2017.

This week, a reader contacted me asking help about executing commands on a remote server. After pointing him to the package [laravelcollective/remote](https://laravelcollective.com/docs/5.3/ssh)³⁶⁷ he was still unclear on how to implement it in his own application, so I decided to create this free tutorial to help him out.

In this tutorial, we will cover:

- installation
- configuration
- running commands on the remote server
- catching output from commands
- creating and running tasks
- SFTP uploads and downloads
- tailing remote logs

Basically, we will cover every bit of functionality mentioned in the package documentation.

Let's get started!

Installation

Currently, in the documentation it says that the latest version is 5.3, but if you visit the repository on [GitHub](https://github.com/LaravelCollective/remote)³⁶⁸ you will see that the latest version is in fact 5.4.

From the command line run:

```
composer require laravelcollective/remote
```

Output:

³⁶⁷ <https://laravelcollective.com/docs/5.3/ssh>

³⁶⁸ <https://github.com/LaravelCollective/remote>

Using SSH to execute commands on a remote server

358

```
$ composer require laravelcollective/remote
Using version ^5.4 for laravelcollective/remote
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing phpseclib/phpseclib (2.0.5)
  Downloading: 100%

- Installing laravelcollective/remote (v5.4.0)
  Downloading: 100%
```

Next, add this provider to the providers in the config/app.php:

```
Collective\Remote\RemoteServiceProvider::class,
```

Finally, in the same file under aliases add this alias:

```
'SSH' => Collective\Remote\RemoteFacade::class,
```

Configuration

To configure this package we first need to publish the configuration file using this command from the command line:

```
php artisan vendor:publish --provider="Collective\Remote\RemoteServiceProvider"
```

A new file is created in config/remote.php.

This file contains these options:

- Default Remote Connection Name - default connection to be used
- Remote Server Connections - array of available connections
- Remote Server Groups - grouped connections

Let's define a new connection in *Remote Server Connections* just below production:

```
'myserver' => [
    'host'      => '',
    'username'  => '',
    'password'  => ''
],
```

You can write your credential directly here, but then your credentials will be visible by everyone who has access to the repository. Smart thing to do is to define the values in your `.env` file and reference them in this file.

Your connection will now look like this:

```
'myserver' => [
    'host'      => env('MYSERVER_HOST'),
    'username'  => env('MYSERVER_USER'),
    'password'  => env('MYSERVER_PASS')
],
```

Now, in your `.env` file add these keys and set the appropriate values:

```
MYSERVER_HOST=Hostname
MYSERVER_USER=Username
MYSERVER_PASS>Password
```

If you need to specify the port other than 22, use `MYSERVER_HOST=YOUR_IP_ADDRESS_HERE:PORT`.

Now, we will change the default connection in `config/remote.php` to `myserver`. When we don't specify the connection to be used, we want to use `myserver`. You can change this to suit your needs.

Running commands on the remote server

To demonstrate running commands we will create a route closure which runs some commands.

Open `routes/web.php` and add this route closure at the end of the file:

```
Route::get('/run', function() {
    \SSH::run([
        'cd /',
        'ls -all'
    ]);

    return response('Completed!');
});
```

You can replace the array of commands with the commands that you want. If you visit the application on /run you will see Completed! when the commands have executed.

If you want to run the commands on a specific connection (in this case production server which is defined in the config/remote.php) you would write it like this:

```
\SSH::into('production')->run([
    'cd /',
    'ls -all'
]);
```

Catching output from commands

In this example, we will display the output from the commands executed on the remote server.

In routes/web.php, add this route closure:

```
Route::get('/output', function() {
    $commands = [
        'cd /',
        'ls -all'
    ];

    \SSH::run($commands, function($line)
    {
        echo $line.PHP_EOL;
    });
});
```

If you visit this route in the browser, you will see the listing of the / directory on your server. Use *View Source Code* in Google Chrome to get prettier output.

Creating and running tasks

You can define a task on a specific connection for commands that are usually executed together.

Let's create a new route closure in routes/web.php, define a task and run it:


```
Route::get('/task', function() {
    \SSH::into('myserver')->define('list', [
        'cd /',
        'ls -all'
    ]);

    \SSH::into('myserver')->task('list', function($line)
    {
        echo $line.PHP_EOL;
    });
});
```

Important! The code above does nothing. I have opened an issue on GitHub. You can view it here [#43³⁶⁹](https://github.com/LaravelCollective/remotelyss/issués/43).

SFTP uploads and downloads

In this chapter we will first upload an empty text file to the server. Then, we will put some text inside that file, retrieve only the text, and finally download the file.

First, create an empty text file in storage/app called sample.txt.

Then, place this code inside routes/web.php:

```
Route::get('/updown', function() {
    // Upload local file `storage/app/sample.txt` to `~/sample.txt` on the server
    \SSH::put(storage_path('app/sample.txt'), 'sample.txt');

    // Place this string inside that file on the server
    \SSH::putString('sample.txt', 'This is sample text.');
```

xt`

```
    // Download the file `~/sample.txt` from the server to `storage/app/sample.t\
    \SSH::get('sample.txt', storage_path('app/sample.txt'));

    // Retrieve the content of file `~/sample.txt` from the server
    $contents = \SSH::getString('sample.txt');

    // display the content of the file
    return response($contents);
});
```

³⁶⁹ <https://github.com/LaravelCollective/remotelyss/issués/43>

After visiting `/updown` in the browser, the previously created empty file `storage/app/sample.txt` will now contain `This is sample text.` string. Also, on the server in the `~/` directory you will see a file `sample.txt` with the same string as in your local file. And in the browser you will also see the content of the file from the server `This is sample text..`

Tailing remote logs

To tail a remote log...

I can't figure out how this works... Maybe it is also broken.

`php artisan tail` works for local `storage/logs/laravel.log` log tailing, but I can't figure out how to make it work on a remote connection.

If you know how it works, let me know in the comments bellow.

This is the current state of the repository at this moment [78699da64ed55cc2aa371648a5d87d5d70c2e241](https://github.com/laravelista/using-ssh-to-execute-commands-on-a-remote-server/commit/78699da64ed55cc2aa371648a5d87d5d70c2e241)³⁷⁰.

Another way of executing commands on the server is by using *Laravel Envoy*. Check out these tutorials to learn more:

- [Deploy Laravel with Envoy](#)
- [Advanced Laravel Envoy](#)

Thank your for reading and have a happy weekend.

³⁷⁰<https://github.com/laravelista/using-ssh-to-execute-commands-on-a-remote-server/commit/78699da64ed55cc2aa371648a5d87d5d70c2e241>

Extras

The most popular tutorial on JWT for Lumen.

JSON Web Token Authentication for Lumen REBOOT

Learn how to implement JWT authentication in your Lumen application the proper way.

Published at: 26. September, 2016.



View Source Code

Source code for this tutorial is available [here](#)³⁷¹.

I see that a lot of you are having problems with my blog post on **JSON Web Token Authentication for Lumen**. I've been reading all comments and emails about it, but because this post was written more than one year ago I decided to reboot the whole post instead of trying to fix issues that have been piling on.

At the time when I wrote that post you could actually follow along and get the wanted result, but packages changed/updated, new versions of Lumen released and things broke.

So, for this tutorial, I will be using Lumen 5.3 (which is the latest Lumen version released at the time of this tutorial) with [tymondesigns/jwt-auth](#)³⁷² package version 1.0.0-alpha.3 to implement JWT authentication in Lumen.

Installation

The starting point for this tutorial is a brand new blank Lumen 5.3 application. This is the current state of the repository at this moment [6f348b603ab6d6c92604ed760e365864da89d0bb](#)³⁷³.

From your Lumen project root directory in the command line type:

```
composer require tymon/jwt-auth:"^1.0@dev"
```

Output:

³⁷¹<https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot>

³⁷²<https://github.com/tymondesigns/jwt-auth>

³⁷³<https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/6f348b603ab6d6c92604ed760e365864da89d0bb>

```
$ composer require tymon/jwt-auth:"^1.0@dev"
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing symfony/polyfill-util (v1.2.0)
  Loading from cache

- Installing symfony/polyfill-php56 (v1.2.0)
  Loading from cache

- Installing namshi/jose (7.2.1)
  Downloading: 100%

- Installing tymon/jwt-auth (1.0.0-alpha.3)
  Downloading: 100%

Writing lock file
Generating autoload files
```

Register the service provider

Now we have to add the service provider to our application. Go to bootstrap/app.php and locate the section *Register Service Providers*. There you will see these commented service providers:

```
// $app->register(App\Providers\AppServiceProvider::class);
// $app->register(App\Providers\AuthServiceProvider::class);
// $app->register(App\Providers\EventServiceProvider::class);
```

Bellow those commented service providers add this uncommented service provider:

```
$app->register(Tymon\JWTAuth\Providers\LumenServiceProvider::class);
```

Copy the config file (optional)

If you need to change something else than the *JWT Authentication Secret*, you can copy the jwt config file to the appropriate directory and make the changes there. Create a new file at config/jwt.php and inside it paste the code from [here](https://raw.githubusercontent.com/tymondsgns/jwt-auth/1.0.0-alpha.3/config/config.php)³⁷⁴.

Generate the secret key

Run the following helper command from the command line:

³⁷⁴<https://raw.githubusercontent.com/tymondsgns/jwt-auth/1.0.0-alpha.3/config/config.php>

```
php artisan jwt:secret
```

My output (your secret key will be different):

```
$ php artisan jwt:secret
jwt-auth secret [hxtjRuBu3qvFQ6Nj7GQyK6au1j55ynTy] set successfully.
```

This command will create a new key/value `JWT_SECRET` in your `.env` file. Before running the command be sure to already have `.env` file created because it will not throw errors if the file is not found.

“This will generate a new random key, which will be used to sign your tokens.” - [source](#)³⁷⁵

This is the current state of the repository at this moment [ffbcdf34a2776f0bf0355b4253b70a6b4803ae7b](#)³⁷⁶.

Integration

We have successfully installed `jwt-auth` at this point. Now we have to integrate it with our Lumen application. To do so we will first instruct Lumen to use the `jwt` driver for the `api` guard. Create a new file at `config/auth.php` and inside it paste the code from [here](#)³⁷⁷.

Before using Lumen’s authentication features, we need to uncomment the call to register the `AuthServiceServiceProvider` service provider in our `bootstrap/app.php` file under *Register Service Providers*.

“The `AuthServiceServiceProvider` located in your `app/Providers` directory contains a single call to `Auth::viaRequest`. The `viaRequest` method accepts a Closure which will be called when the incoming request needs to be authenticated. Within this Closure, you may resolve your `App\User` instance however you wish. If no authenticated user can be found for the request, the Closure should return null.” - [source: Lumen Authentication](#)³⁷⁸

In `app/Providers/AuthServiceProvider`, replace the existing `viaRequest` with the following:

```
$this->app['auth']->viaRequest('api', function ($request) {
    if ($request->input('email')) {
        return User::where('email', $request->input('email'))->first();
    }
});
```

And finally, we need to implement the `JWTSubject` interface in our `User` model. Go to our user model located in `app/User.php` and at the top add the use statement `use Tymon\JWTAuth\Contracts\JWTSubject;`. Then append `JWTSubject` to the class *implements* list. Finally, we need to implement methods that the `JWTSubject` interface requires. Add these two methods to the `User` class:

³⁷⁵<https://github.com/tymondesigns/jwt-auth/wiki/Installation>

³⁷⁶<https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/ffbcdf34a2776f0bf0355b4253b70a6b4803ae7b>

³⁷⁷<https://gist.github.com/mabasic/7979d67ce3ec75a5938e3d14575736a6/raw/61d1e5d49a450c3aae2289ef4c55c900e99180b6/auth.php>

³⁷⁸<https://lumen.laravel.com/docs/5.3/authentication>

```

/**
 * Get the identifier that will be stored in the subject claim of the JWT.
 *
 * @return mixed
 */
public function getJWTIdentifier()
{
    return $this->getKey();
}

/**
 * Return a key value array, containing any custom claims to be added to the JWT.
 *
 * @return array
 */
public function getJWTCustomClaims()
{
    return [];
}

```

You User class should now look something like [this](#)³⁷⁹.

One more thing before we continue and it is important. In your `.env` file set `CACHE_DRIVER=file`. This will avoid the Class 'Memcached' not found error.

Authentication

There are two things that I will show you how to do in this chapter:

1. Log in using email and password to obtain a JWT
2. Use obtained JWT to access a protected route

Getting the token

Before we can do the actual login, we need to setup our database, write a migration file for the users table and seed our initial user.

Database setup

In your `.env` file add/change the key `DB_CONNECTION` to `sqlite`.

Create empty database file called `database.sqlite` in database directory. Then create a new migration using:

³⁷⁹ <https://gist.github.com/mabasic/9fb7f910ed073eeb0a6c5f2ba5bf9c89>

```
$ php artisan make:migration create_users_table --create=users
Created Migration: 2016_09_26_113558_create_users_table
```

Your migration file name will be different, based on your current date & time. Open that migration file and make the up method look like this:

```
Schema::create('users', function (Blueprint $table) {
    $table->increments('id');
    $table->string('name');
    $table->string('email')->unique();
    $table->string('password');
    $table->timestamps();
});
```

Next we need to modify the database/factories/ModelFactory.php file a bit to reflect our User model:

```
$factory->define(App\User::class, function (Faker\Generator $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->email,
        'password' => app('hash')->make('testtest')
    ];
});
```

And finally we need to seed a user to our database. Go to database/seeds/DatabaseSeeder.php file and inside the run method add this code:

```
factory(App\User::class)->create([
    'name' => 'Tester',
    'email' => 'test@test.com',
    'password' => app('hash')->make('testtest')
]);
```

Now let's migrate & seed the database with this command:

```
$ php artisan migrate --seed
Migration table created successfully.
Migrated: 2016_09_26_113558_create_users_table
```

That should do it.

This is the current state of the repository at this moment [6495bcecd8e92a79aebdc530776edb8e37643951](https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/6495bcecd8e92a79aebdc530776edb8e37643951)³⁸⁰.

³⁸⁰<https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/6495bcecd8e92a79aebdc530776edb8e37643951>

Login setup

Before we proceed, let's talk about verifying that the application works. I personally use [Postman](https://www.getpostman.com/)³⁸¹ while working with APIs. Just visit their website and install the appropriate software.

One quick note about using Postman to test your API. When using PUT, PATCH methods be sure to use `x-www-form-urlencoded` option in Postman instead of default `form-data` when sending data in a request.

Big thanks to Michael Dyrynda on [larachat](https://larachat.co/)³⁸² for helping me solve this issue.

To keep things simple, we will add the login logic to our routes file located in `routes/web.php`:

```
use Illuminate\Http\Request;
use Tymon\JWTAuth\JWTAuth;

$app->post('login', function(Request $request, JWTAuth $jwt) {
    $this->validate($request, [
        'email' => 'required|email|exists:users',
        'password' => 'required|string'
    ]);

    if (! $token = $jwt->attempt($request->only(['email', 'password']))) {
        return response()->json(['user_not_found'], 404);
    }

    return response()->json(compact('token'));
});
```

If you haven't already, start up your Lumen application with `php -S localhost:8000 -t public/`.

Use Postman to POST to `/login` URL and set data for email `test@test.com` and password `testtest`. When you send that data you will be greeted with a token:

³⁸¹<https://www.getpostman.com/>

³⁸²<https://larachat.co/>

```
{
  "token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOi8vbG9jYXRob3N\
00jgwMDAvbG9naW4iLCJpYXQiojE0NzQ4OTE2MTIsImV4cCI6MTQ3NDg5NTIxMiwibmJmIjoxNDc0ODk\
xNjEyLCJqdGkiOiJhYTBiMDU0NTI0YjVkJmJlE3ZTc1ZGZlZWJmFkNjAyNiIsInN1YiI6MX0.ywXvcf5\
16_DS3A5yc6Z3uD4EpEZY8n8I8PXIOwnH9TE"
}
```

Your token will be different. You will have to remember (write down) the token that you have received because we will need it for making requests that require authentication.

This is the current state of the repository at this moment [700345f85902b0520f6da6a76cacd88d46f7e434](https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/700345f85902b0520f6da6a76cacd88d46f7e434)³⁸³.

Accessing protected routes

In order to be able to use the parts of the API that require authentication, you will have to set an authorization header on our HTTP request (Postman) as follows:

Authorization: Bearer {yourtokenhere}

or you can include the token via a query string, for example:

`http://api.mysite.com/admin?token={yourtokenhere}`

Before we create the protected route, there are a few more things that we have to do to:

1. Register auth middleware
2. Enable eloquent

Register auth middleware

Since our route group uses middleware auth to check that the user has provided the token, we have to register that middleware. To do so, go to `bootstrap/app.php` under *Register Middleware* section and uncomment `$routeMiddleware` like so:

```
$app->routeMiddleware([
    'auth' => App\Http\Middleware\Authenticate::class,
]);
```

Enable eloquent

Authentication won't work without enabling eloquent. In `bootstrap/app.php` file, uncomment the line `$app->withEloquent();`.

We will now create a protected route that will return the currently logged in user. Go to `routes/web.php` and paste the following code at the end of the file:

³⁸³ <https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/700345f85902b0520f6da6a76cacd88d46f7e434>

```
$app->group(['middleware' => 'auth'], function () use ($app) {
    $app->post('user', function (JWTAuth $jwt) {
        $user = $jwt->parseToken()->toUser();
        return $user;
    });
});
```

As you can see, we are using a route group with auth middleware assigned. Inside the group, we have a POST route for /user in which we use dependency injection to inject JWTAuth. Then using JWTAuth we retrieve the logged in user and return it.

This is the current state of the repository at this moment [e911fce0f3615f745e93ae98e662e15a95e94bc5](https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/e911fce0f3615f745e93ae98e662e15a95e94bc5)³⁸⁴.

Global exception handling

Since tokens can expire or be invalid, we will hook up into app/Exceptions/Handler.php file render method to watch for those exceptions and return appropriate JSON responses.

```
public function render($request, Exception $e)
{
    if ($e instanceof \Tyron\JWTAuth\Exceptions\TokenExpiredException) {
        return response()->json(['token_expired'], 500);
    } else if ($e instanceof \Tyron\JWTAuth\Exceptions\TokenInvalidException) {
        return response()->json(['token_invalid'], 500);
    }

    return parent::render($request, $e);
}
```

This is the current state of the repository at this moment [e511a6c3eed4f1bf16affad54c5e69c339e5f178](https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/e511a6c3eed4f1bf16affad54c5e69c339e5f178)³⁸⁵.

This concludes this tutorial. I have shown you how to install jwt-auth in Lumen 5.3, configure it, protect routes, login, make requests that require authentication and retrieve the authenticated user. You can view the whole repository at GitHub. Have fun and don't forget to star if you like it.

There are now a few more things that I would like to talk about.

³⁸⁴<https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/e911fce0f3615f745e93ae98e662e15a95e94bc5>

³⁸⁵<https://github.com/laravelista/json-web-token-authentication-for-lumen-reboot/commit/e511a6c3eed4f1bf16affad54c5e69c339e5f178>

The future of jwt-auth for Lumen

It took me two attempts to write this tutorial; that is to get jwt-auth working with Lumen 5.3. In my first attempt I tried to use the original 0.5.9 version of the package and with the help of the comments you guys posted on the original post, I managed to install and configure the package but got stuck on authentication (getting the token). I tried all workarounds that I could think of (without major code changes) and nothing.

Then I noticed that the jwt-auth package has an alpha release with the LumenServiceProvider. A quick look around the repository [issues](#)³⁸⁶ and I've found a way to get it working with that release.

Now about the future of the package. The comments on the issues page seem lively enough and people are willing to help. The documentation is one year old and feels that it needs attention. Maybe that will change with the 1.0.0 release. I must admit that after trying the alpha release and seeing how good it integrates with Lumen, this package is going places. It has the potential to be good and in my opinion, the Lumen integration could only get better, not worse from this point.

After some quick search on [Packalyst](#)³⁸⁷ and [Packagist](#)³⁸⁸, I have found that only this package is available for JWT authentication for Laravel/Lumen.

Do you really need JWT?

This thought has been reoccurring in my mind during the whole time I was writing this tutorial. It was not easy to install this package and get JWT authentication working. I don't see a way that a beginner could have easily installed this package without being very persistent. And to make things worse, there are only a few PHP packages on Packagist that work with JWT.

Maybe I am mistaking, but to me, it feels like too much work for such a simple thing. Laravel since version 5.2 already comes with API authentication out of the box and in 5.3 they have improved it even more. In my original post about this subject, I had a chapter in which I reflected on my experiences while building the same API with Lumen and Laravel. I am sticking with my original opinion.

If you need a full blown API you are better off with Laravel. If you need a simple API that does a thing or two, you will be better off with Lumen, but if you get to the point where you need JWT in Lumen you are probably off track and doing too much in it.

Thank you for reading this tutorial.

³⁸⁶ <https://github.com/tymondesigns/jwt-auth/issues/513>

³⁸⁷ <http://packalyst.com/>

³⁸⁸ <https://packagist.org/>

What is a canonical tag?

If you have the same content on multiple URLs then the canonical tag is something that you should know about and use it.

Published at: 09. July, 2017.

While working on [Visit Murter](https://visitmurter.com)³⁸⁹ I realized that I display almost the same content on two different URLs. You can see here on [Destinations - Murter - Beaches - Slanica](https://visitmurter.com/destinations/murter/explore/beaches/slanica)³⁹⁰ that the content is almost the same as is in [Explore - Beaches - Slanica](https://visitmurter.com/explore/beaches/slanica)³⁹¹.

The reason why we need the same content on both URLs is because of the way that the users come to that content.

The first URL is related to the destination Murter and its beaches:

`https://visitmurter.com/destinations/murter/explore/beaches/slanica`

The users come to this page by first picking a destination on the island and then exploring it.

While the second URL is related to all destinations on the island Murter and its beaches:

`https://visitmurter.com/explore/beaches/slanica`

The users come to this page by first choosing to explore everything on the island regardless of the destination.

The difference between those two sites is in the provided navigation. On the destinations Murter page, the navigation provides you with links to everything that you can see/do on that destination (Murter, Betina, Jezera, Tisno, NP Kornati), while on the explore page the navigation provides you with links to everything that you can see/do on the whole island.

I Hope that this makes sense to you.

So, where does the canonical tag come into play?

³⁸⁹<https://visitmurter.com>

³⁹⁰<https://visitmurter.com/destinations/murter/explore/beaches/slanica>

³⁹¹<https://visitmurter.com/explore/beaches/slanica>

The canonical tag

The canonical tag tells search engines which version of the URL you want to appear in the search results.

Having same content on different URLs is bad because search engines see that the same content is on both URLs and then they rank only one of those URLs. By setting the canonical tag you are in control of which URL is to be displayed in the search results.

Using canonicalization helps you control your duplicate content.

I bet that you did not know this

Search crawlers might be able to reach your homepage in all of the following ways:

- <http://www.laravelista.com>
- <https://www.laravelista.com>
- <http://laravelista.com>
- <http://laravelista.com/index.php>
- <http://laravelista.com/index.php?refer=twitter>

To a human being, all of the above URLs represent a single page. To a search crawler, every one of the above URLs is a unique page. By setting the canonical tag to a specific URL (master copy of a page) we are telling search engines that all other URLs are a copy (duplicate) of the page provided in the canonical tag.

In the case above, setting the canonical tag to let's say <http://laravelista.com> will increase the SEO ranking for that URL by 50%. It's like you are merging the SEO ranking for those URLs into one.

How do I apply this knowledge to my website

The first step is ultra easy.

1. Canonicalize your home page

Add this HTML meta tag to you head section:

What is a canonical tag?

375

```
<link rel="canonical" href="https://yourdomain.goeshere" />
```

Replace `https://yourdomain.goeshere` with your actual domain and protocol that you want to rank for on search engines. This will have huge benefits for your website rankings.

1. Examine if you have duplicate content and place the canonical tag accordingly

If you have like me same content on two URLs you can place the canonical tag on the page which you don't want to be ranked (duplicate), to point the page that you want to rank (master copy). Also, you can place the canonical tag on the page that you want to rank (master copy) and point it to that same page. *Both parts are important.*

Example

Your **master copy page** URL is `http://domain.com/content-a` and your **duplicate page** URL is `http://domain.com/content-b`. On the master copy page you should place this canonical tag:

```
<link rel="canonical" href="http://domain.com/content-a" />
```

And on your duplicate page/s you should place this canonical tag:

```
<link rel="canonical" href="http://domain.com/content-a" />
```

URL `http://domain.com/content-a` will be ranked, and the ranking from `http://domain.com/content-b` will go towards `http://domain.com/content-a`.

Thank you for reading. Have a great weekend!

Sources

- [What is Canonicalization?](#)³⁹²

³⁹²<https://moz.com/learn/seo/canonicalization>

Loading your own fork of a third party library

If you are using a certain library for your project and you decide to change something in the library, you will want your project to use the patched version.

Published at: 30. May, 2017.

A few days ago, I received an email from a reader asking me how to change something in a package that I created. I told him that currently, it is not possible to do what he requested, but that he can fork the package, make the change and use that modified version in his application by leveraging Composer.

This is a common case scenario and it is well described in the Composer documentation: [Loading a package from a VCS repository](#)³⁹³.

In this tutorial I will show you how to fork a package ([laravelista/comments](#)³⁹⁴), make changes to the package and use that modified version in your Laravel application.

Forking

First things first, navigate to the repository of the package that you want to make changes on. For the purpose of this tutorial, this is the package [laravelista/comments](#)³⁹⁵.

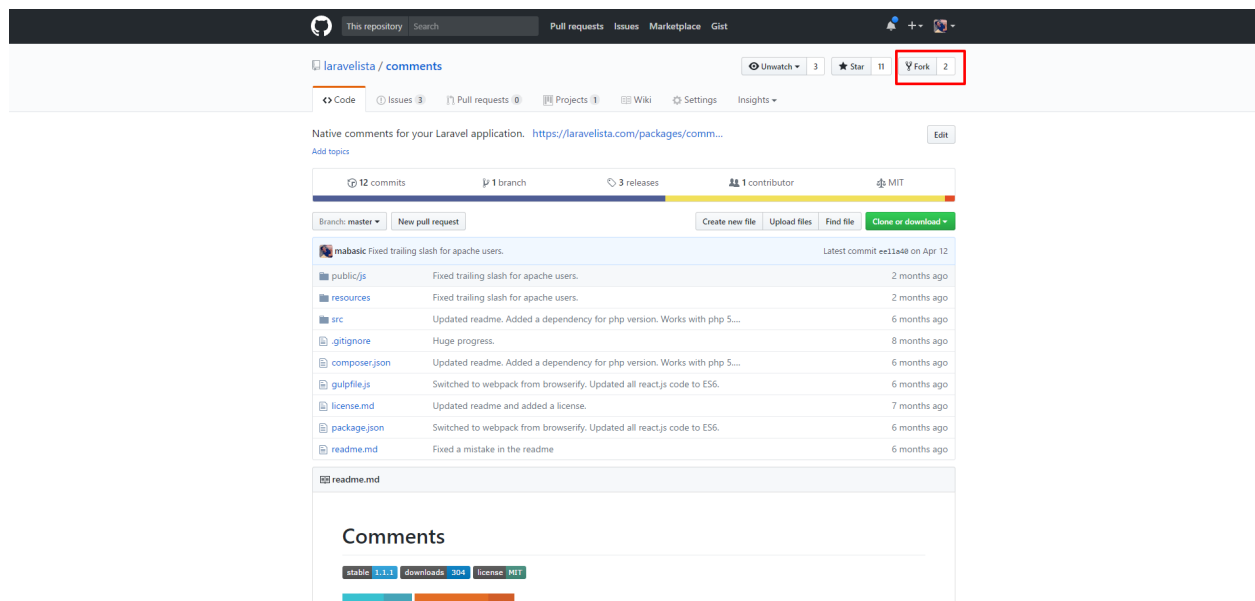
You have to have a GitHub account for this operation and you have to be logged in.

You will see a *Fork* button in the upper right corner of the page.

³⁹³<https://getcomposer.org/doc/05-repositories.md#loading-a-package-from-a-vcs-repository>

³⁹⁴<https://github.com/laravelista/comments>

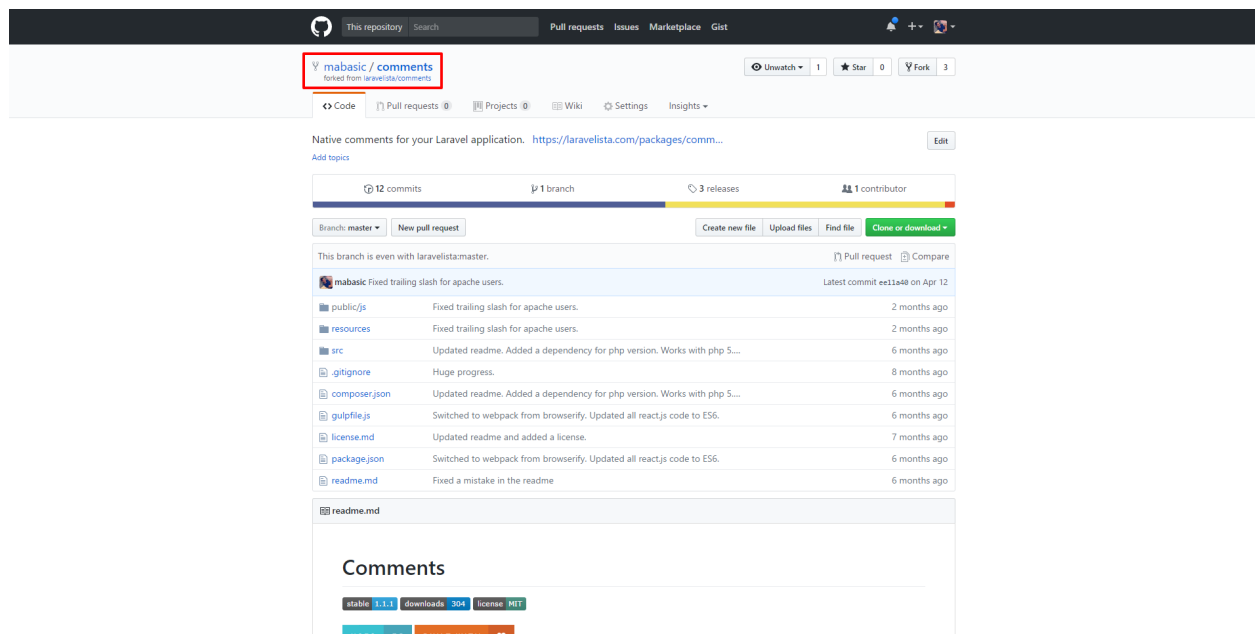
³⁹⁵<https://github.com/laravelista/comments>



fork

Click on it. It will ask you where do you want to fork it (to which account/team). I will fork it to my own personal GitHub profile [@mabasic³⁹⁶](#).

As you can see in the image bellow the repository is now forked to my personal account:



Forked

Now we can proceed to make needed changes to the repository.

³⁹⁶<https://github.com/mabasic>

Changing

You can make changes to the repository on GitHub using your browser or you can clone the repository to your PC and make the changes there.

```
git clone git@github.com:mabasic/comments.git comments-fork  
  
cd comments-fork
```

I made changes to the `CommentList.jsx` and `CommentForm.jsx` files. For this example, I translated the text when there are no comments and when the user is not logged in, to Croatian.

Now for this package to compile the JS code we need to execute:

```
npm install  
  
gulp
```

This will create everything that we need.

Commit the changes and push to GitHub. You can view the changes [here](#)³⁹⁷.

Requiring

We are ready now to use our fork of the package in our Laravel application.

One thing to remember here is that we made the changes in the *master* branch. Usually, you would want to make the changes in a different branch or release a new version.

In your Laravel application open `composer.json` file and add the package to the require section:

```
"require": {  
    ...  
    "laravelista/comments": "dev-master"  
},
```

To instruct Composer to use our forked version instead, add this above the require section:

³⁹⁷ <https://github.com/mabasic/comments/commit/2f1ef8a883c1d1b28b510aa0804946981d4fb428>

```
"repositories": [
    {
        "type": "vcs",
        "url": "https://github.com/mabasic/comments"
    }
],
```

When you run `composer update` now, you should get something similar to this:

```
$ composer update
Loading composer repositories with package information
Updating dependencies (including require-dev)
- Installing laravelista/syndra (1.2.1)
  Loading from cache

- Installing league/fractal (0.14.0)
  Loading from cache

- Installing laravelista/comments (dev-master 2f1ef8a)
  Downloading: 100%
```

Notice the part **Installing laravelista/comments (dev-master 2f1ef8a)** The last string is the code of the commit that we just pushed to GitHub. You can verify that by going to the [commits³⁹⁸](https://github.com/mabasic/comments/commits/master) page for the package.

To further verify that we got the forked package, you can visit the `vendor/laravelista/comments` directory and manually check if the files that we made changes on are located there.

Thank you for reading this tutorial. I am sure that you will find it very useful. Have an awesome week!

³⁹⁸<https://github.com/mabasic/comments/commits/master>

The End

Thank you for reading!

Tell me your opinions about this book at mario@laravelista.hr³⁹⁹.

³⁹⁹ <mailto:mario@laravelista.hr>